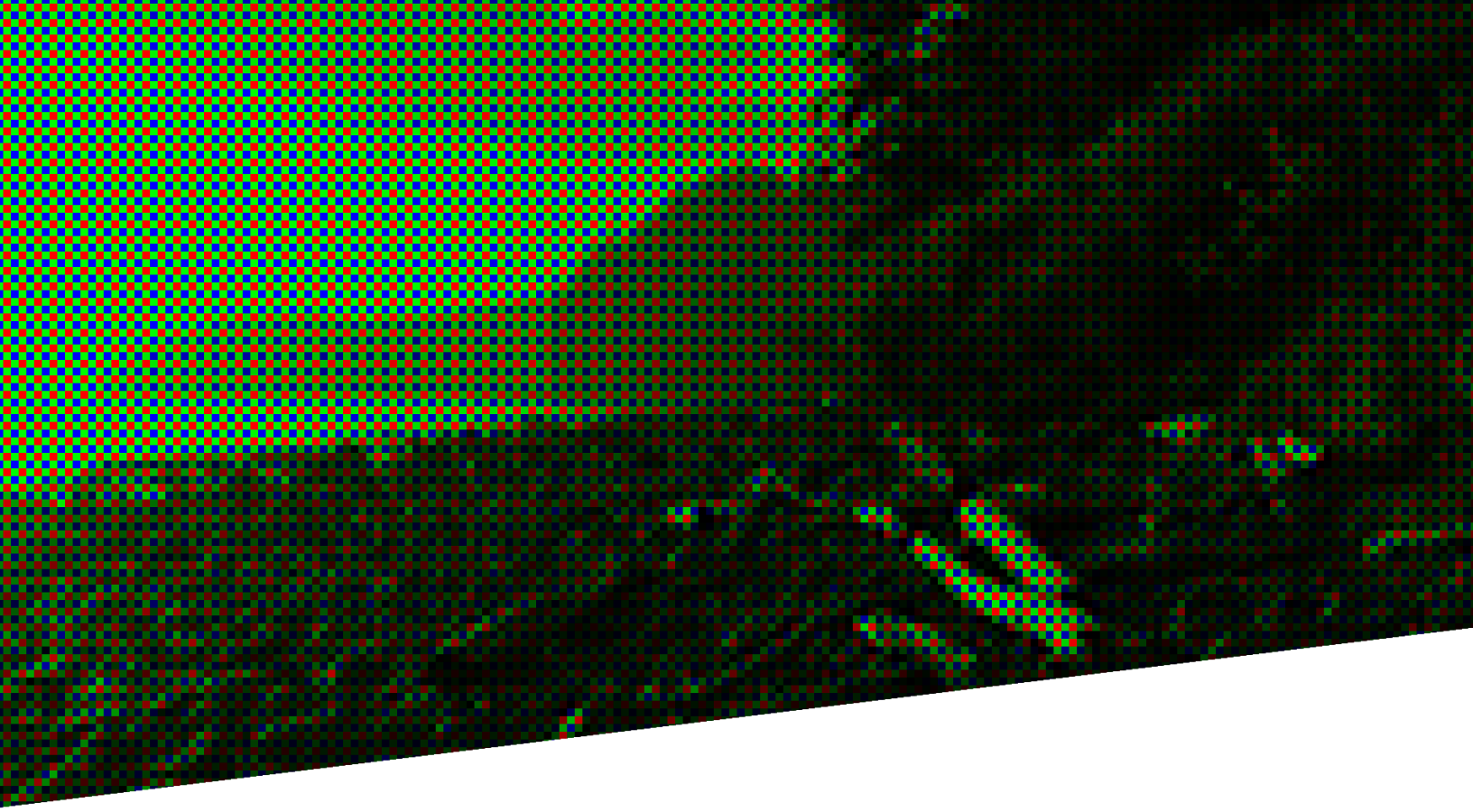


University of Victoria
Faculty of Engineering

SENG 440 A01 – Final Report
August 17, 2023
Revision 0

Optimization Techniques for Image Demosaicing Through Bilinear Interpolation

Curtis Upshall
curtisupshall@uvic.ca
V00899873

Kjartan Einarsson
kjartaneinarsson@uvic.ca
V00885049

Contents

Glossary

1. **RGB Color Space**: A color model in which colors are represented as combinations of three primary colors - red, green, and blue. It is widely used in digital imaging and displays to represent a wide range of colors.
2. **Photosite**: Also known as a pixel, it is the smallest unit of a digital image sensor that captures light and converts it into an electrical signal. Multiple photosites collectively form an image.
3. **LCD (Liquid Crystal Display)**: A flat-panel display technology that uses liquid crystals to modulate light and produce images. LCDs are commonly used in monitors, TVs, and other electronic devices.
4. **Integrated Circuit (IC)**: A miniaturized electronic circuit that consists of interconnected semiconductor components, such as transistors, diodes, and resistors, on a single chip of semiconductor material.
5. **Moiré**: An undesirable interference pattern that occurs when two regular patterns, such as grids or lines, are overlaid or closely aligned, causing an irregular, wavy appearance.
6. **Signal-to-Noise Ratio (SNR)**: A measure of the strength of a signal relative to the background noise. In various applications, including imaging and electronics, a higher SNR indicates a clearer and more reliable signal.
7. **ARM NEON**: A SIMD (Single Instruction, Multiple Data) architecture extension for ARM processors. NEON technology enables processors to perform parallel processing on multiple data elements, enhancing performance in multimedia and signal processing applications.

Abstract

Image demosaicing is a vital process for real-time image rendering and video compression. This report presents the implementation of a real-time algorithm for image demosaicing using bilinear interpolation. We explore the significance of demosaicing in modern digital cameras, particularly CMOS sensors utilizing the Bayer filter. The report details the algorithm's design, challenges, hardware solutions, and software optimizations.

The Bayer filter, commonly used in CMOS sensors, presents a challenge for recovering color image details, due to its mosaic arrangement of photosites. Bilinear interpolation, a simple yet effective technique, is employed for its ability to handle this pattern. Our demosaicing algorithm is designed to process 24-bit pixels using 32-bit kernels, facilitating efficient memory access.

Software optimizations are investigated, including loop fusion, partial redundancy elimination, and constant propagation. These optimizations enhance the algorithm's performance, reducing execution time and improving cache utilization.

The report provides insights into the trade-offs between simplicity and performance in real-time image processing. The proposed algorithm, implemented and tested on ARM architecture, serves as a foundation for further hardware acceleration and optimization efforts.

1. Introduction

Image demosaicing (also known as image debayering, named after the Bayer filter) is a process used to interpolate data within an image that has had a mosaic filter applied. The most common application of this technique in engineering contexts is for demosaicing CMOS sensor images, which predominantly use the Bayer filter, shown in Figure 1. Today, nearly all commercially sold digital cameras employ CMOS sensors to capture images, and thus depend upon a demosaicing algorithm of some kind.

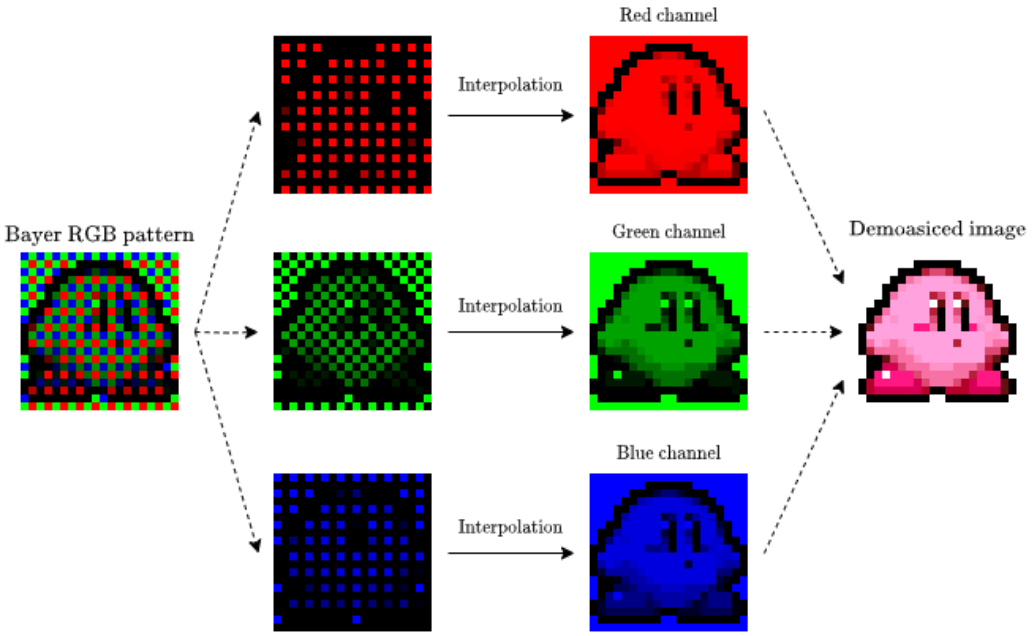


Figure 1: Image demosaicing process.

In many cases such as RAW-format digital photography, image information is saved as a mosaic, and demosaicing is performed during image rendering. However, there are common scenarios where demosaicing must be performed in real time. For example, one of the most common video compression techniques involves converting from the RGB color space into a different color space (for example, the YCC color space) that more efficiently conveys visual information. Because this process cannot work on image frames that have a mosaic filter applied, a demosaicing algorithm must interpolate missing information in the image in real time before the image frame can be compressed.

This report details the implementation of a real time algorithm for bilinear interpolation used to achieve image demosaicing. The next section contextualizes the need for image demosaicing and illustrates some algorithms of particular interest. Following sections of this report detail the implementation of these algorithms.

2. Background

One of the most common tools for displaying digital images is the LCD, which employs picture elements - or "pixels" - that are arranged into squares consisting of a red, green and blue sub-pixel, as shown in Figure 2 (left). Each sub-pixel typically has 8 bits of precision, thus providing 255 (2^8) different shades for each sub-pixel, and 24 bits of precision per pixel. On a macroscopic scale, these three channels can produce over 16.5 million (255^3) colors, allowing humans to perceive full-color images. While LCD sub-pixels are commonly arranged into rectangles that form square pixels, engineers avoid using rectangular photo sites on CMOS sensors, since this decreases pixel pitch - the density between photosites on a sensor. Because pixel pitch positively correlates with the perception of photons [2], engineers arrange photosites into a mosaic of square photosites to increase pixel pitch. Although other mosaic patterns (see Figure 3) provide similar visual performance, the Bayer pattern is the most popular, as it compensates for the fact that the human eye's increased distribution of green-sensitive cone cells. Consequently, loss of detail in the green channel of an image is more noticeable (see Figure 4), so Bayer filters provide twice as many green pixels as red or blue pixels.

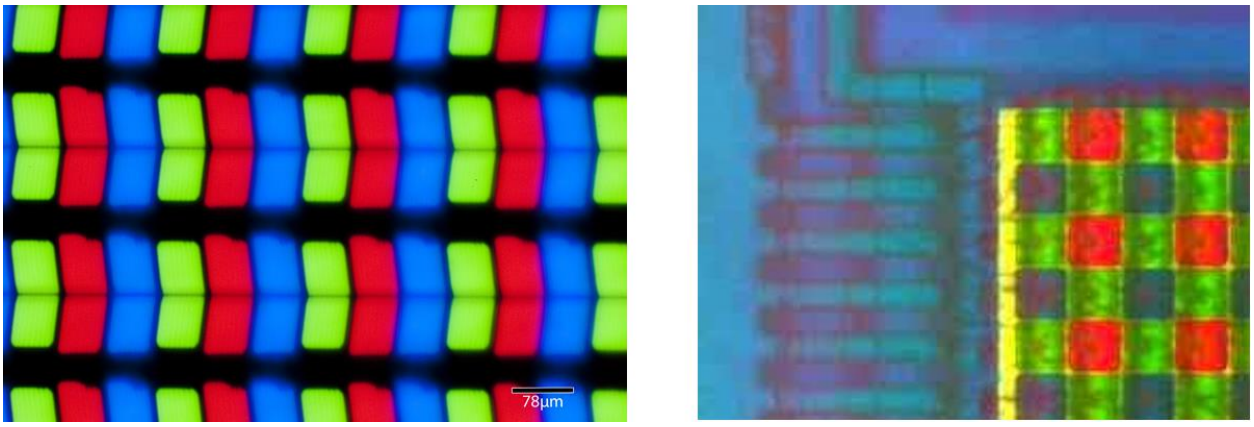


Figure 2: Display pixels vs. sensor photosites. Left: LCD on a Retina iPad. Sub-pixels are arranged into vertical strips that comprise square pixels [1]. Right: CMOS camera sensor. Photosites are square and arranged into a Bayer pattern.

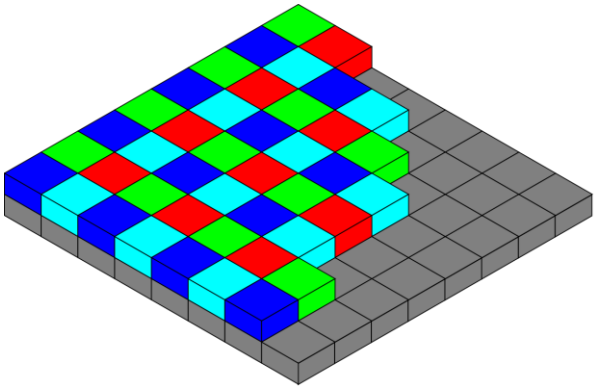


Figure 3: The RGBE filter, an alternative to the Bayer filter. It uses a fourth color: "emerald" (commonly referred to as cyan). [3]

Figure 4: Demonstration of the spectral sensitivity of the human eye. In each image, one of the RGB channels has been blurred, simulating a loss of detail. Notably, loss of clarity in the green channel has the greatest effect on the image, even for parts of the image that aren't green. This issue is especially prevalent for human skin tones. Left: Blue channel loss. Center: Green channel loss. Right: Red channel loss.

2.1. Interpolation

While not limited to image manipulation applications, interpolation is a widely used technique in digital imaging, primarily used to resample images. While there are various methods available for interpolation (see Figure 5), we favor bilinear interpolation for its simplicity and ease of implementation. In the context of mathematics, linear interpolation is a method of constructing new data within a discrete set of known values by inferring a linear relationship. By extension, bilinear interpolation is a process for interpolating a function of two variables using repeated linear interpolation.

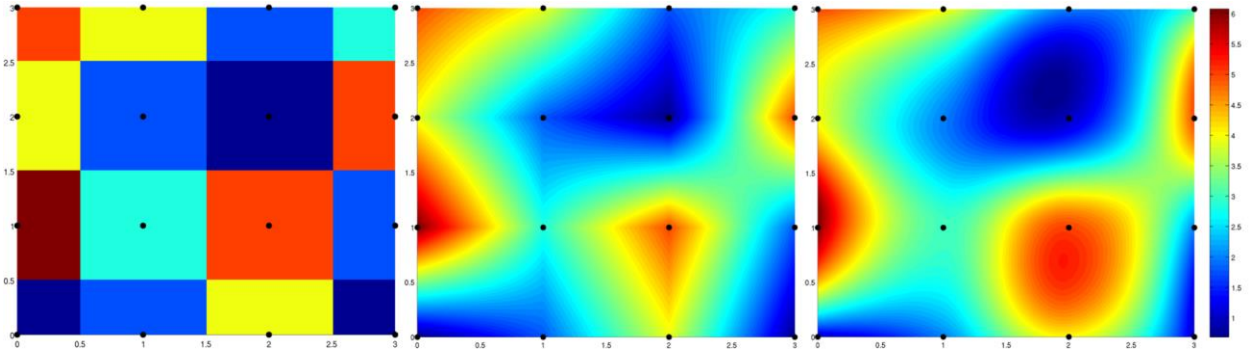


Figure 5: Different methods of interpolation [4]. Left: Nearest neighbor. Center: Bilinear. Right: Bicubic.

For an unknown function $y = f(x)$, the value y of the function for a particular input x can be linearly interpolated using two known input and output pairs, (x_0, y_0) and (x_1, y_1), by the following relation:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}. \tag{1}$$

Solving this relation for y, we have:

$$y = \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0}. \tag{2}$$

Trivially, a single row of pixels in a digital image is essentially represented by pixel luminance, y, as a function of space, x. However, digital images of interest are typically 2-dimensional. In this case, we extend linear interpolation using bilinear interpolation, namely by performing linear interpolation in the x-direction, and then again in the y-direction.

For the application of demosaicing, let us consider an image to be an unknown function f of two variables, x and y. Suppose we wish to find the value of f and a point (x, y). It is assumed that we know the value of f at four points, namely $Q_{11} = (x_1, y_1), Q_{12} = (x_1, y_2), Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$. First we apply linear interpolation in the x-direction, which yields:

$$f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \tag{3}$$

$$f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \tag{4}$$

Next, we interpolate in the y-direction. For completeness, we show the complete expression representing this operation:

$$y(x, y) = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \tag{5}$$

From Equation 3 and 4, given that $x_2 - x_1$ is equivalent to 2 for imputing over a single pixel, the imputed value for a particular sub-pixel is exactly equal to the sum of each of its neighboring pixel's sub-pixels divided by 2. This process is illustrated later in the report.

3. Design Challenges

As with all image and video processing applications, image demosaicing offers many challenges for engineers to overcome. Because demosaicing stands as a prerequisite for video compression, it's important that our solution performs adequately in real time scenarios. To that end, there are numerous aspects of numerical accuracy to consider. Firstly, inaccuracies from arithmetic rounding should not adversely affect image quality. Secondly, the system should introduce a minimal amount of noise during demosaic in order to preserve the signal-to-noise ratio of the image. Thirdly, global image artifacts like moiré should be avoided, and edge details should be preserved as well as possible.

4. Hardware Solutions

In most engineering contexts today, real time image processing is typically performed by a graphics processing unit or GPU. GPUs commonly feature dedicated hardware for performing tasks like demosaicing, color space conversion, image scaling, and many other techniques. By the same token, we apply dedicated hardware to perform demosaicing on an image in order to bolster software performance. As discussed in the previous section, bilinear interpolation involves inputting first in the x-direction, then

in the y-direction. Consequently, we employ two types of integrated circuits (ICs), namely row imputers and column imputers. These ICs are illustrated in Figure 6 and 7.

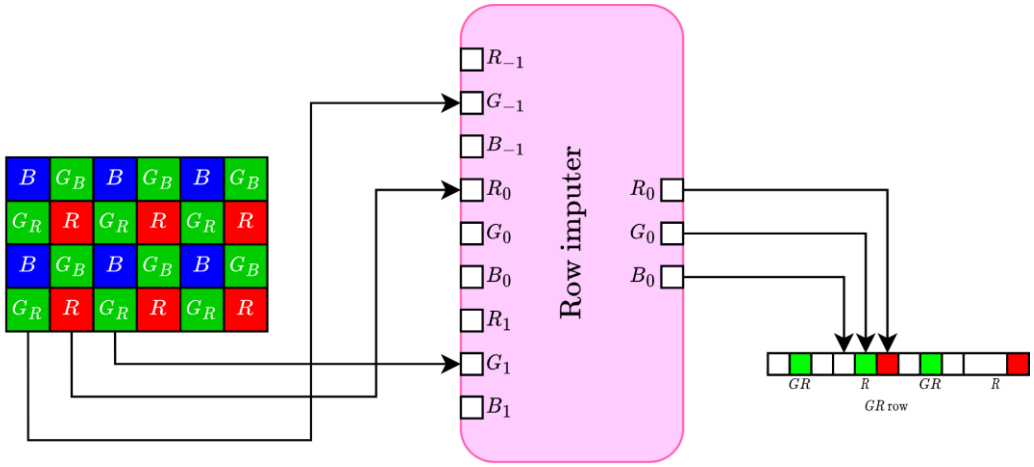


Figure 6: An IC performs linear interpolation of an 8-bit image. The green channel of the first R pixel in the first GR row is imputed by the IC. Here, R_{-1} refers to the red channel of the previous pixel, R_0 being the currently interpolated pixel, and R_1 being the next pixel in the image; This applies to the G and B inputs also. Pixels are written to a 24-bit color image buffer.

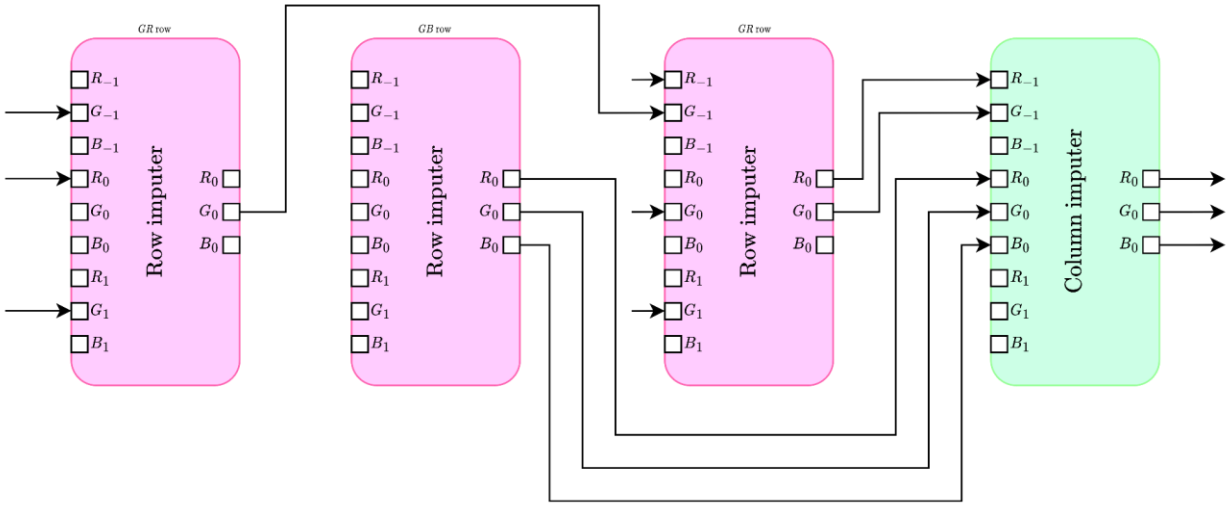


Figure 7: As extension of the row imputer, the column imputer linear interpolates pixel columns. Picture here are multiple row imputers that are daisy-chained together.

While the elaborate engineering effort required to realize this hardware design is outside of the scope of this project, the potential optimization would be significant; It currently takes up to 6 instructions to impute the value of a single sub-pixel, 4 of which are dedicated to bit manipulation within a kernel and may be replaced by a single imputation instruction. The code used to impute a red sub-pixel in **K0_0** is shown below in Figure 8.

```
// 1. Read K0_1[R0] into p0_r
tmp0_r = p0_r;
p0_r = k0_1;
p0_r = p0_r & 0x0000FF00;
p0_r = p0_r >> 8;
tmp0_r = tmp0_r + p0_r; // Combine p0_r
// 1. Read K0_4[G0] into p0_gb
tmp0_gb = p0_gb;
p0_gb = k0_4;
p0_gb = p0_gb & 0x000000FF;
tmp0_gb = tmp0_gb + p0_gb; // Combine p0_gb
// 2. Write p0_r to K0_0[R0].
tmp0_r = tmp0_r >> 1; // Divide by 2
tmp0_r = tmp0_r << 16;
k0_0 = k0_0 & 0xFF00FFFF;
k0_0 = k0_0 | tmp0_r;
// 2. Read K0_4[B1] into p0_b
tmp0_b = p0_b;
p0_b = k0_4;
p0_b = p0_b & 0x00FF0000;
p0_b = p0_b >> 16;
tmp0_b = tmp0_b + p0_b; // Combine p0_b
// 3. Advance K0_0.
pixels[y * ROW_SIZE + (3 * x)] = k0_0; // Write K0_0 back to memory
k0_0 = pixels[y * ROW_SIZE + (3 * x) + 3]; // Read into K0_0
```

Figure 8: Bits are shifted and masked to interpolate sub-pixel values spanning two kernels.

5. System Specifications

In this report, we consider our solution to be the system that implements the demosaicing algorithm. We choose to target Bayer filter images; Notably, the Bayer consists of red-green rows and blue-green rows. We label red and green pixels from the red-green row as R pixels and GR pixels respectively, and we label blue and green pixels from the blue-green row as B pixels and GB pixels respectively, as shown in Figure 9. Our solution reads 24-bit encoded bitmap (BMP) mosaic images using the Bitmap V5 header with Windows encoding, and writes demosaiced images in the same format. We choose Bitmap image encoding, as it allows us to easily address individual sub-pixels within an image. Our solution accepts bitmap images that are 4000 pixels × 6000 pixels, as we wanted to process images large enough to accurately assess cache performance.

In conjunction with our demosaic algorithm, we also implement an additional program that encodes color images into mosaiced images. Converting 24-bit color bitmaps to 24-bit mosaiced bitmaps trivializes the image conversion process; Typically, a mosaiced image would be a grayscale 8-bit encoded bitmap, with each pixel (namely R, B, GR, or GB) being inferred from its position. Retaining our input bitmaps as 24-bit essentially inserts “bubbles” in the image during the encoding process, which our demosaicing algorithm subsequently fills in. This is illustrated in Figure 10. For the purpose of this report, performance metrics for the encoding algorithm are not considered.

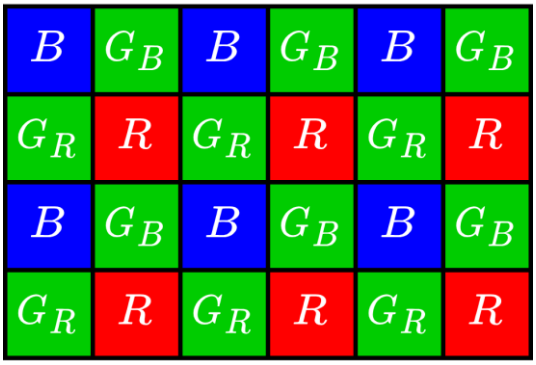


Figure 9: Bayer filter with labeled pixels.

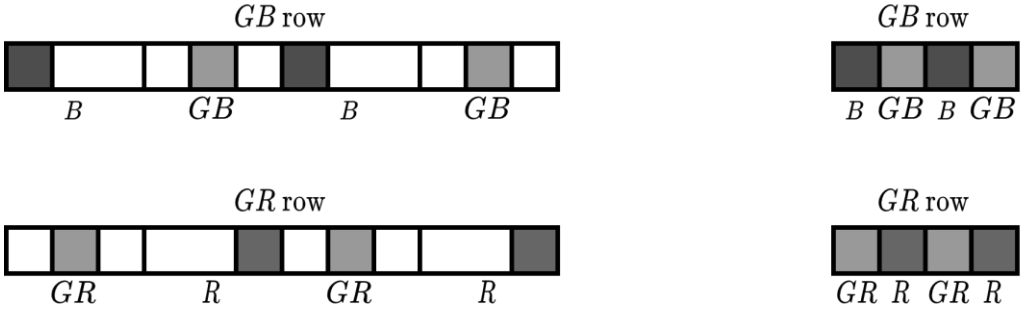


Figure 10: 24-bit vs 8-bit bitmap mosaic image pixels. Left: 24-bit bitmap. For GR and GB pixels, the green channel is occupied while the remaining two channels are zeroed, making the pixel appear green when the mosaic image is rendered. The same principle applies for B and R pixels. Right: 8-bit bitmap. The image contains no color data; Color imagery is only realized when each pixel type is inferred during demosaicing.

5.1. Processor & Compilation

Our solution was written in C and compiled by ARM GCC for the ARM920T processor, which uses the ARM v4 architecture. The code was developed and tested on a FriendlyARM Mini2440 development board, accessed over SSH by the University of Victoria's Electrical Engineering Department. The CPU architecture supports both 32-bit and 16-bit Thumb instructions in one instruction set. For the purposes of this report, benchmarking of our final solution utilizes a QEMU ARM simulator. Our final solution was compiled at optimization level 3 (using the **-O3** compiler flag) as well as the ARM **-mfpu=neon** compilation flag. The results of these compilation flags are discussed in the Optimizations section of this report.

6. System Design

Because the CPU architecture we are targeting has a 32-bit word size, we process image pixels in 32-bit kernels. Since each image pixel uses 24-bits of precision, we can process 4 pixels using 3 registers, as shown in Figure 11. These kernels are labeled **K0_0** through **K_02** for green-red rows, and **K0_3** through **K_05** for blue-green rows.

The algorithm first imputes pixel colors row-wise, then column-wise. Because pixel values for a given channel are only interpolated over a distance of one pixel, the algorithm simply sums the value of its two neighboring pixels for a given channel, then divides this value in half. Consequently, the demosaicing algorithm uses fixed-point arithmetic with a fixed-point scale factor of 1. Considering rounding, the value of a particular sub-pixel belongs in the range $[0, 255]$, and may be off by a value of due to integer rounding. Our algorithm employs two rows of red-green kernels and two rows of blue-green kernels at a time, in order to expose parallelism to the compiler; The process is shown below in Figure 12. Notably, the CPU architecture uses little endian encoding, so byte orders of each sub-pixel within a kernel are in reverse order.

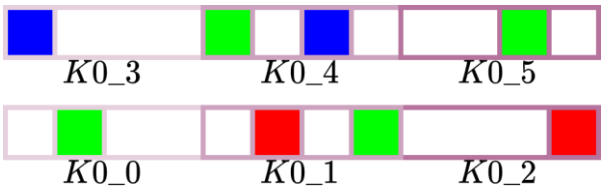


Figure 11: Image pixel kernels.

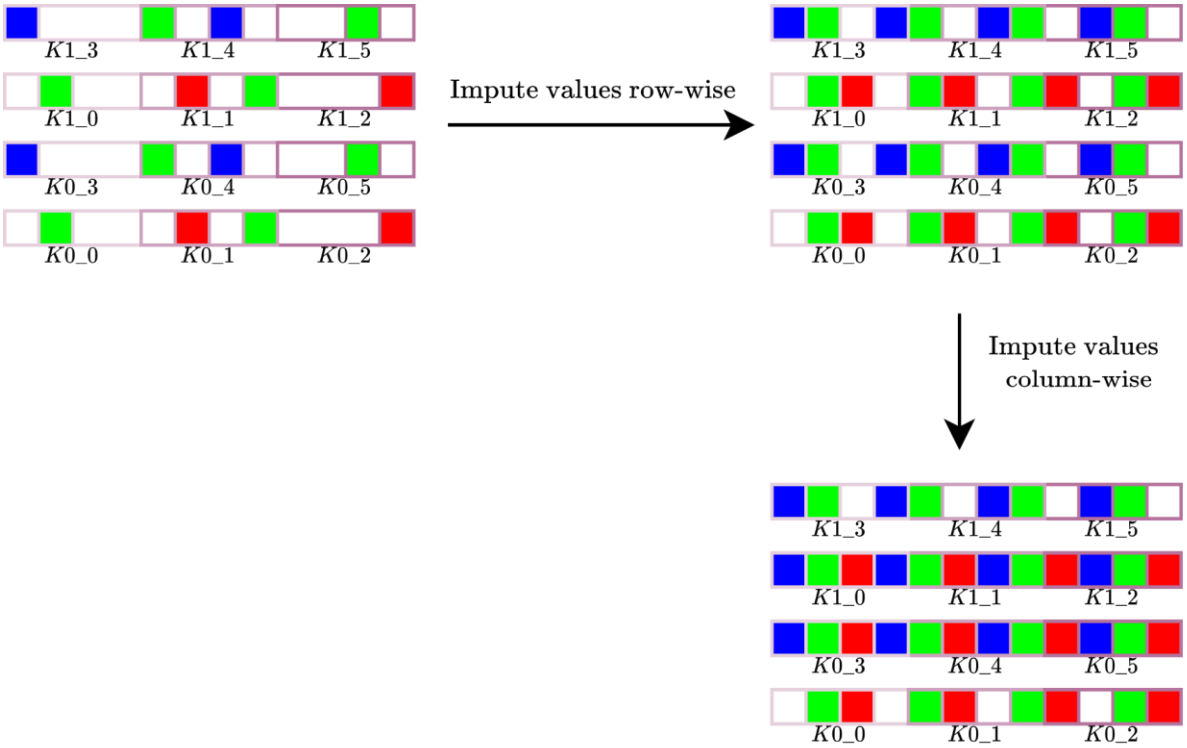


Figure 12: Order of operations for the demosaic algorithm.

6.1. System Limitations

Because linearly interpolating an unknown function assumes a linear relationship, image interpolation using this method can sometimes result in color artifacting, especially around areas of sharp contrast as

shown in Figure 13. While this limitation does have an effect on the performance of our system, this issue is much less prevalent for larger images. This issue could also be ameliorated by sampling two or more points during the linear interpolation phase of the algorithm without needing to change our interpolation strategy.

As a consequence of addressing pixels in groups of 4, the input images of our system must have pixel dimensions such that its width is divisible by 12 (4 pixels × 3 kernels) and its height is divisible by 6. Although this initially seems like a very limiting constraint, demosaicing algorithms in practice typically work with very large images of a fixed size. Furthermore, images decoded by our algorithm contain noise along the bottom, right and top edge, since the algorithm does not have data outside the boundaries of the image to impute. We acknowledge this as a limitation of our solution; In practical engineering applications, this would likely be solved in hardware.

Figure 13: Image artifacts from bilinear interpolation. Left: original image. Right: image has been encoded and then decoded. Color artifacting is only visible around areas of sharp contrast.

7. Software Optimizations

This section details the various software optimization techniques considered for our solution. To that end, there were several techniques that we attempted to implement that either yielded little to no benefit, or were too complex to implement for our system. Among the optimizations that proved little effectiveness were constant folding, loop unrolling and operator strength reduction. Loop unrolling seemed to have little effect on the runtime performance of our algorithm, so we elected to remove it from our final solution. Constant folding was also attempted, but the compiler would produce one or two fewer instructions when we didn't attempt constant folding, so we removed this optimization. Furthermore, operator strength reduction appeared to be automatically optimized by the compiler.

Although our final solution was compiled using level 3 optimization, we also compiled using levels 0 through 2 to form a basis of comparison for compiler optimizations. These results are shown in Table 1.

Optimization Level	*Assembly Code (LOC)*
O0	2096
O1	813
O2	794
O3	794

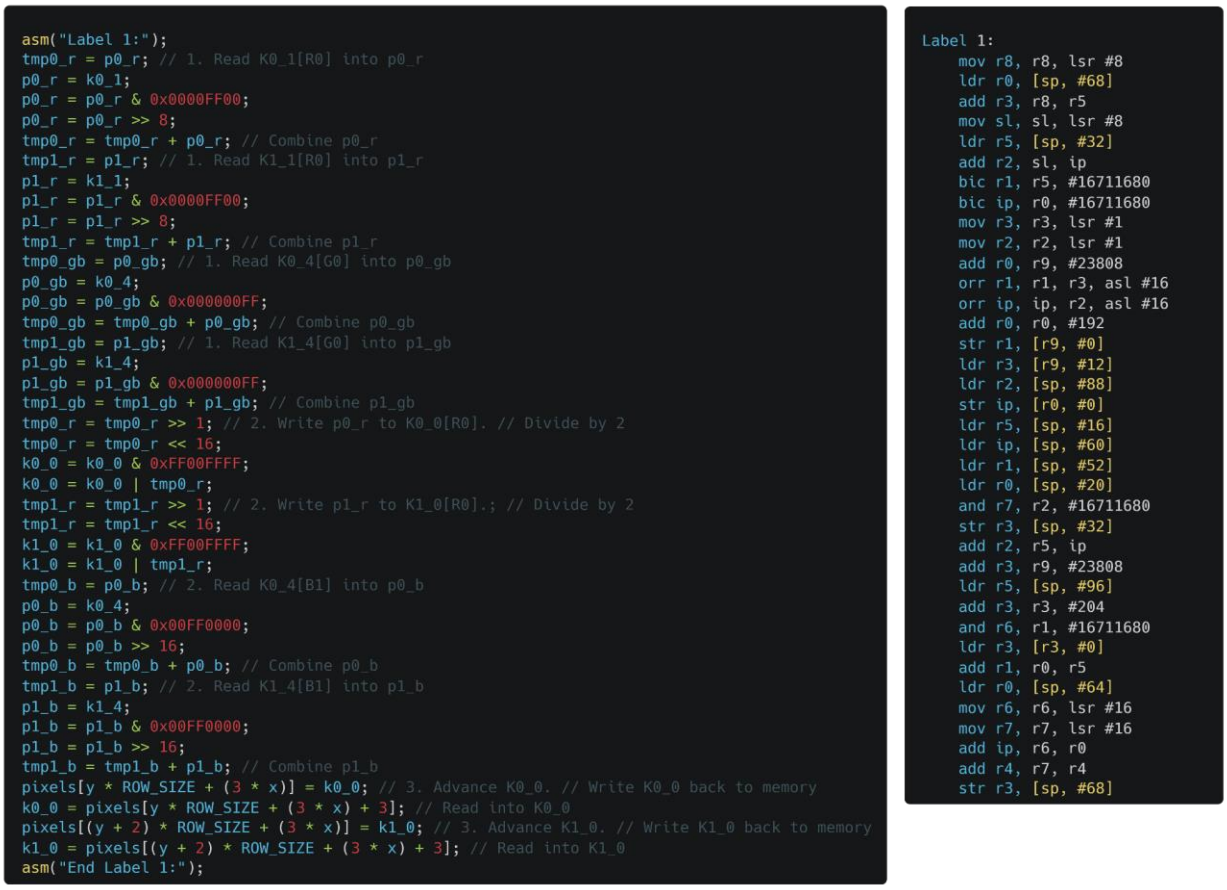
Table 1: Optimization level results

7.1. Optimization 1: Software Pipelining & Loop Fusion

Our primary optimization strategy involved exposing parallelism to the compiler through loop fusion. By operating on two rows of kernels at the same time, the compiler is able to vastly reduce the total number of instructions for a given kernel interpolation procedure. The resulting assembly code is shown in Figure 14. At optimization level 3, the ARM GCC will attempt to automatically vectorize any operations that it can; the **-mfpu=neon** does the same thing, but for ARM NEON operations. Since optimization level 2 yielded the same amount of assembly code, we can conclude that the compiler could not automatically vectorize these operations. On a two-slot machine, this operation stands to offer a significant speedup.

7.2. Optimization 2: Partial Redundancy Elimination

Partial redundancy elimination was commonly used to optimize interpolation subroutines. This technique was especially prevalent for optimizing out unnecessary bit manipulation operations, as shown below in Figure 15.



```
asm("Label 1:");
tmp0_r = p0_r; // 1. Read K0_1[R0] into p0_r
p0_r = k0_1;
p0_r = p0_r & 0x0000FF00;
p0_r = p0_r >> 8;
tmp0_r = tmp0_r + p0_r; // Combine p0_r
tmp1_r = p1_r; // 1. Read K1_1[R0] into p1_r
p1_r = k1_1;
p1_r = p1_r & 0x0000FF00;
p1_r = p1_r >> 8;
tmp1_r = tmp1_r + p1_r; // Combine p1_r
tmp0_gb = p0_gb; // 1. Read K0_4[G0] into p0_gb
p0_gb = k0_4;
p0_gb = p0_gb & 0x000000FF;
tmp0_gb = tmp0_gb + p0_gb; // Combine p0_gb
tmp1_gb = p1_gb; // 1. Read K1_4[G0] into p1_gb
p1_gb = k1_4;
p1_gb = p1_gb & 0x000000FF;
tmp1_gb = tmp1_gb + p1_gb; // Combine p1_gb
tmp0_r = tmp0_r >> 1; // 2. Write p0_r to K0_0[R0]. // Divide by 2
tmp0_r = tmp0_r << 16;
k0_0 = k0_0 & 0xFF00FFFF;
k0_0 = k0_0 | tmp0_r;
tmp1_r = tmp1_r >> 1; // 2. Write p1_r to K1_0[R0].; // Divide by 2
tmp1_r = tmp1_r << 16;
k1_0 = k1_0 & 0xFF00FFFF;
k1_0 = k1_0 | tmp1_r;
tmp0_b = p0_b; // 2. Read K0_4[B1] into p0_b
p0_b = k0_4;
p0_b = p0_b & 0x00FF0000;
p0_b = p0_b >> 16;
tmp0_b = tmp0_b + p0_b; // Combine p0_b
tmp1_b = p1_b; // 2. Read K1_4[B1] into p1_b
p1_b = k1_4;
p1_b = p1_b & 0x00FF0000;
p1_b = p1_b >> 16;
tmp1_b = tmp1_b + p1_b; // Combine p1_b
pixels[y * ROW_SIZE + (3 * x)] = k0_0; // 3. Advance K0_0. // Write K0_0 back to memory
k0_0 = pixels[y * ROW_SIZE + (3 * x) + 3]; // Read into K0_0
pixels[(y + 2) * ROW_SIZE + (3 * x)] = k1_0; // 3. Advance K1_0. // Write K1_0 back to memory
k1_0 = pixels[(y + 2) * ROW_SIZE + (3 * x) + 3]; // Read into K1_0
asm("End Label 1:");
```

```
Label 1:
    mov r8, r8, lsr #8
    ldr r0, [sp, #68]
    add r3, r8, r5
    mov sl, sl, lsr #8
    ldr r5, [sp, #32]
    add r2, sl, ip
    bic r1, r5, #16711680
    bic ip, r0, #16711680
    mov r3, r3, lsr #1
    mov r2, r2, lsr #1
    add r0, r9, #23808
    orr r1, r1, r3, asl #16
    orr ip, ip, r2, asl #16
    add r0, r0, #192
    str r1, [r9, #0]
    ldr r3, [r9, #12]
    ldr r2, [sp, #88]
    str ip, [r0, #0]
    ldr r5, [sp, #16]
    ldr ip, [sp, #60]
    ldr r1, [sp, #52]
    ldr r0, [sp, #20]
    and r7, r2, #16711680
    str r3, [sp, #32]
    add r2, r5, ip
    add r3, r9, #23808
    ldr r5, [sp, #96]
    add r3, r3, #204
    and r6, r1, #16711680
    ldr r3, [r3, #0]
    add r1, r0, r5
    ldr r0, [sp, #64]
    mov r6, r6, lsr #16
    mov r7, r7, lsr #16
    add ip, r6, r0
    add r4, r7, r4
    str r3, [sp, #68]
```

Figure 14: Demonstration of loop fusion. Kernel 0_0 and 1_0 are operated on within the same loop, effectively interpolating two rows of GR pixels concurrently. Notably, there are fewer lines of assembly than there are of C code. Left: C code. Right: Assembly code.

```
label1: ; Before
    mov r2, r2, lsr #1
    bic r3, r3, #65280
    mov r1, r1, lsr #1
    bic r0, r0, #65280
    bic r5, ip, #255
    bic fp, r4, #255

label2: ; After
    ldr r4, [sp, #4]
    mov r2, r2, lsr #1
    mov r1, r1, lsr #1
    orr r3, r4, r2, asl #8
    orr r0, r0, r1, asl #8
```

Figure 15: Partial redundancy elimination, before and after.

7.3. Optimization 3: Constant Propagation

Because our algorithm targets a particular bitmap image size, we elected to use static constant variables for image width, height and row byte size. This brought our solution down from 855 LOC to 804 LOC.

7.4. Optimizations: Future Work

Having explored a wide gamut of optimization techniques, some methods of optimization could not be implemented due to time constraints and engineering complexity.

7.4.1. SIMD & NEON Intrinsics

ARM NEON intrinsics allow engineers to directly interface with ARM's Single-Input-Multiple-Data (SIMD) architecture. While this architecture can be invaluable for image processing applications, the complexities of implementing NEON intrinsics were too great for the scope of this project. One of the major challenges of implementing NEON intrinsics involved loading Bitmap subpixel values into the 128-bit NEON registers.

8. Results & Discussion

Benchmarking our solution involved running 4 versions of our decoding algorithm across 8 bitmap files, each version containing further software optimizations. Gprof was used to assess timing performance of each solution version. The latter half of this section details the results of each optimization version of the solution.

While the final version uses 32-bit kernels for processing images, our first implementation of the demosaic algorithm (hereafter, Version 1) comprised a simple nested loop, iterating over each sub-pixel and interpolating both row-wise and column-wise simultaneously. Counter to our expectations on cache performance, this initial implementation outperformed our latest solution, even for very large images that should not fit in the cache. Although we have included the performance metrics of this solution in our report, we focused our optimization efforts on the kernel implementation. The reason for this is twofold; Firstly, the engineering effort to produce the kernel solution was much higher than we had anticipated, depleting us of any remaining time to explore alternative solutions. Secondly, the kernel solution (hereafter, Version 2) remains very prevalent in the realm of online algorithms, whereby data of unknown length is streamed in a single pass. Given that our project focuses primarily on real time applications of demosaicing in the context of video processing, we decided to focus solely on this approach. Moreover, we found that the kernel solution was more grounded in the design of our proposed hardware solution.

We initially developed our program to run for a single image, resulting in Gprof timing metrics that were too small to validate any optimization efforts. For this reason, we ran our benchmarks on all 8 sample images.

8.1. Software Version 1

Version 1 was the first working iteration of the demosaic algorithm. Notably it finishes execution that fastest, at 1.20 seconds:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
39.17	0.47	0.47				rowStrt
36.67	0.91	0.44				ColStrt
8.33	1.01	0.10				read
6.67	1.09	0.08				write
5.00	1.15	0.06				munmap
4.17	1.20	0.05				open
0.00	1.20	0.00	8	0.00	0.00	decodeImage
0.00	1.20	0.00	8	0.00	0.00	writeImage
0.00	1.20	0.00	1	0.00	0.00	main

% time the percentage of the total running time of the program used by this function.

8.2. Software Version 2

Version 2 utilized the aforementioned kernel configuration as set out by our system design. This improved the algorithm's memory access efficiency; To that end, we use 6 kernels of pixel data in each loop. Notably, this solution exposes parallelism to the compiler. The solution finishes execution in 1.61 seconds:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
40.99	0.66	0.66				ColStrt
39.13	1.29	0.63				RowStrt
6.83	1.40	0.11				write
4.97	1.48	0.08				read
3.73	1.54	0.06				open
2.48	1.58	0.04				munmap
1.24	1.60	0.02				_wordcopy_fwd_dest_aligned
0.62	1.61	0.01				_IO_file_finish
0.00	1.61	0.00	8	0.00	0.00	decodeImage
0.00	1.61	0.00	8	0.00	0.00	loadImage
0.00	1.61	0.00	8	0.00	0.00	writeImage
0.00	1.61	0.00	1	0.00	0.00	main

% time the percentage of the total running time of the program used by this function.

8.3. Software Version 3

Version 3 further refined the system design and removed unused memory access requests and unnecessary bit manipulation operations. It finishes execution in 1.54 seconds:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
43.51	0.67	0.67				ColStrt
35.71	1.22	0.55				rowStrt
7.14	1.33	0.11				write
5.84	1.42	0.09				read
4.55	1.49	0.07				open
1.95	1.52	0.03				munmap
0.65	1.53	0.01				_wordcopy_fwd_dest_aligned
0.65	1.54	0.01				memcpy
0.00	1.54	0.00	8	0.00	0.00	decodeImage
0.00	1.54	0.00	8	0.00	0.00	loadImage
0.00	1.54	0.00	8	0.00	0.00	writeImage
0.00	1.54	0.00	1	0.00	0.00	main

% time — the percentage of the total running time of the program used by this function.

8.4. Software Version 4

Version 4 implored the use of constant propagation, eliminating the memory access and space held for the original variable. It finished executing in 1.53 seconds:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
42.48	0.65	0.65				ColStrt
38.56	1.24	0.59				rowStrt
6.54	1.34	0.10				write
5.88	1.43	0.09				read
2.61	1.47	0.04				open
1.96	1.50	0.03				munmap
0.65	1.51	0.01				memcpy
0.65	1.52	0.01				mmap
0.65	1.53	0.01				moncontrol
0.00	1.53	0.00	8	0.00	0.00	decodeImage
0.00	1.53	0.00	8	0.00	0.00	loadImage
0.00	1.53	0.00	8	0.00	0.00	writeImage
0.00	1.53	0.00	1	0.00	0.00	main

% time — the percentage of the total running time of the program used by this function.

9. Conclusion

This report features some of the many optimization techniques we explored while implementing a single-pass linear interpolation algorithm used for image demosaicing. Image demosaicing stands as an important problem in digital imaging, namely in CMOS sensors employing the Bayer filter. To that end, we have discussed at length the significance of Bayer filters and their importance in engineering contexts.

Our results and discussions highlighted the evolution of our solution through different software versions, each iteration bringing us closer to a more optimized and efficient demosaicing algorithm. We recognized the challenges of balancing memory access efficiency, parallelism, and computational complexity. While our project provides a solid foundation for real-time image demosaicing, there is still ample room for further exploration and improvement. Future work could include the integration of ARM NEON intrinsics for enhanced SIMD parallelism, as well as more advanced hardware acceleration techniques. Additionally, investigating other interpolation methods and exploring their trade-offs could contribute to a more comprehensive understanding of real-time image processing solutions.

In conclusion, this report has documented our efforts in designing, implementing, and optimizing a real-time image demosaicing algorithm. Through a series of iterations and optimizations, we have demonstrated the potential for achieving efficient image processing in the context of modern digital imaging applications. Our work serves as a steppingstone for future endeavors in the realm of hardware-accelerated demosaicing and beyond.

10. References

[1] *Apple Retina Display*, Bryan Jones. June 24, 2010. Article.
Available: https://prometheus.med.utah.edu/~bwjones/2010/06/apple-retina-display/

[2] *What is Pixel Pitch and Why Does It Matter?* Planar. February 23, 2018. Blog post.
Available: https://www.planar.com/blog/2018/2/23/what-is-pixel-pitch-and-why-does-it-matter/

[3] *RGBE filter*. Wikipedia. Available: https://en.wikipedia.org/wiki/RGBE_filter

[4] *Interpolation.* Wikipedia. Available: https://en.wikipedia.org/wiki/Interpolation