CPE 442 - Tutorial 6

Justin Carlson
Benjamin Lappen

December 11, 2024

1 Optimization Process

To start, we both re-ran our Tutorial 5 code with O2, O3, and Ofast flags to compare to our non-optimized (O0) performance. On the Raspberry Pi 4, we interestingly saw little performance uplift by using the flags. However, with the Raspberry Pi 5, O2 and O0 showed negligible differences in performance, running a 1080p 30FPS video at about 17 FPS. However, with the O3 flag, performance massively increases, running at 26 FPS, a 9 FPS uplift. Ofast improves performance very slightly on the Pi 5, running at 27 FPS overall. Looking into the perf event counters, the cache misses are nearly the same regardless of optimization level. Toggling the O3 flag does result in a large reduction in instructions run throughout the program, resulting in the large performance uplift seen.

Beyond toggling optimization flags we additionally spent time looking through our code to catch unoptimal routines and improve upon them.

2 Convolution Inner Loop Unrolling

We decided to unroll our main sobel calculation loop so instead of applying the sobel kernel on every index in the 3x3 matrix individually for an eight pixel vector, we apply this operation per pixel on each of the three rows of the sobel kernel which allowed us to find a decent speed up. Here's what the original loop looks like:

```
for (int ki = -1; ki <= 1; ki++) {
    for (int kj = -1; kj <= 1; kj++) {
        // get BGR value for the pixel position + the convolution offset
        uint8x8_t anchor_vector = vld1_u8(&image.ptr<uchar>(row + ki)[j + kj]);

        // cast uint8x8 to int16x8 in order for calculations to work
        int16x8_t anchor_vector_int = vreinterpretq_s16_u16(vmovl_u8(anchor_vector));

        // duplicate current Gx filter coefficient into all elements of int16x8
        int16x8_t Gx_coeff = vdupq_n_s16(Gx[ki + 1][kj + 1]);

        // duplicate current Gy filter coefficient into all elements of int16x8
        int16x8_t Gy_coeff = vdupq_n_s16(Gy[ki + 1][kj + 1]);

        // vector multiply the pixels by Gx coefficients and accumulate
        gradX = vmlaq_s16(gradX, anchor_vector_int, Gx_coeff);

        // vector multiply the pixels by Gy coefficients and accumulate
        gradY = vmlaq_s16(gradY, anchor_vector_int, Gy_coeff);
    }
}
```

This is what the -1 row of the three new sobel calculations look like:

```
const uchar *row_minus_1 = image.ptr<uchar>(i - 1); {
    // get each vector of the top row of image values for 8 consecutive pixels
    uint8x8_t vec_left = vld1_u8(&row_minus_1[j - 1]);
    uint8x8_t vec_center = vld1_u8(&row_minus_1[j]);
    uint8x8_t vec_right = vld1_u8(&row_minus_1[j + 1]);

    // cast from u8 to i16 vector
    int16x8_t vec_left_int = vreinterpretq_s16_u16(vmovl_u8(vec_left));
    int16x8_t vec_center_int = vreinterpretq_s16_u16(vmovl_u8(vec_center));
    int16x8_t vec_right_int = vreinterpretq_s16_u16(vmovl_u8(vec_right));

    // multiply by kernel value and accumulate to gradient
    gradX = vmlaq_n_s16(gradX, vec_left_int, -1);
    gradX = vmlaq_n_s16(gradX, vec_right_int, 1);

    gradY = vmlaq_n_s16(gradY, vec_left_int, 1);
    gradY = vmlaq_n_s16(gradY, vec_center_int, 2);
    gradY = vmlaq_n_s16(gradY, vec_right_int, 1);
}
```

The center row leverages a special performance benefit from the middle row of the G_y kernel being all zeros.

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Since the unrolled loop operates over this entire middle row at once by multiplying everything by zero, we can completely ignore the G_y calculation which offers a decent speed-up. This is what the middle row operation looks like without the redundant calculation:

```
const uchar *row_center = image.ptr<uchar>(i); {
    // get each vector of the top row of image values for 8 consecutive pixels
    uint8x8_t vec_left = vld1_u8(&row_center[j - 1]);
    uint8x8_t vec_right = vld1_u8(&row_center[j + 1]);

    // cast from u8 to i16 vector
    int16x8_t vec_left_int = vreinterpretq_s16_u16(vmovl_u8(vec_left));
    int16x8_t vec_right_int = vreinterpretq_s16_u16(vmovl_u8(vec_right));

    // multiply by kernel value and accumulate to gradient
    gradX = vmlaq_n_s16(gradX, vec_left_int, -2);
    gradX = vmlaq_n_s16(gradX, vec_right_int, 2);
}
```

With this improvement, we were able to gain between a 5-10% performance increase between the convolution unrolling and by removing the y-gradient calculation.

3 Greyscale Memory Access Improvements

After plenty of experimentation, we found that our program made too many expensive calls to OpenCV's Mat::ptr inside our greyscale function. Our original method was the following where we calculated the correct pixel offset manually by calling ptr for every pixel vector individually.

```
for (int i = 0; i < image.rows; i++) {
    for (int j = 0; j < image.cols; j += 8) {
        // calculate data offset from current position/channel
```

```
        int index = i * image.step + j * channels;

        // load 24 bytes of data representing 8 pixels into a uint8 8x3 matrix
        uint8x8x3_t rgb_vec = vld3_u8(&image.data[index]);
        ...
        // store all 8 grey pixels into their respective locations
        vst1_u8(&grey_image.ptr<uchar>(i)[j], grey_u8);
```

For reference, here's what a single call to ptr looks like:

```
template<typename _Tp> inline
_Tp* Mat::ptr(int y)
{
    CV_DbgAssert( y == 0 || (data && dims >= 1 && (unsigned)y < (unsigned)size.p[0]) );
    return (_Tp*)(data + step.p[0] * y);
}
```

There seems to be a number of dereferences to Mat class attributes along with a lot of arithmetic/logical instructions and a pointer cast that could add up over a large number of calls.

Rather than call this for every pixel, we can call this once per row and increment the pointer every time we need to write to a new chunk of eight pixels. Here is the code replacing the beginning of our greyscale function:

```
for (int i = 0; i < image.rows; i++) {

    uchar *image_row = image.ptr<uchar>(i);
    uchar *grey_row = grey_image.ptr<uchar>(i);

    for (int j = 0; j < image.cols; j += 8) {

        // load 24 bytes of data representing 8 pixels into a uint8 8x3 matrix
        uint8x8x3_t rgb_vec = vld3_u8(&image_row[j * channels]);
        ...
        // store all 8 grey pixels into their respective locations
        vst1_u8(&grey_row[j], grey_u8);
```

Creating the pointer ahead of time had a noticeable impact on our total instruction count and gave a surprising 10-15% improvement by itself as we were able to get rid of the redundant assert and other operations. We also implemented a similar mechanic in our convolution function that had a similar pattern, but given the overhead of the rest of the function it didn't provide the same speedup as it did in greyscale.

4 Conclusion

Testing for this project was done on a Pi 4. Our tutorial5 code ran our benchmark at 34.6 frames per second on a 60 frames per second video. The two performance improvements on tutorial6 bumped this up to 42.7 frames per second, offering about a 23% increase overall without any optimization flags. Compiling with -o3 produced results roughly identical to -o0. Testing the new changes with -ofast gave a slightly lower output of 42.2 frames per second.