Optimizing code to darken a bitmap, part 3

devblogs.microsoft.com/oldnewthing/20220309-00

March 9, 2022

Raymond Chen

Our investigation into a simple function to darken a bitmap is still trying to beat this function:

```
union Pixel
{
    uint8_t c[4]; // four channels: red, green, blue, alpha
    uint32_t v;   // full pixel value as a 32-bit integer
};

void darken(Pixel* first, Pixel* last, int darkness)
{
  int lightness = 256 - darkness;
  for (; first < last; ++first) {
    first->c[0] = (uint8_t)(first->c[0] * lightness / 256);
    first->c[1] = (uint8_t)(first->c[1] * lightness / 256);
    first->c[2] = (uint8_t)(first->c[2] * lightness / 256);
  }
}
```

We transformed the problem into subtracting off the darkness rather than preserving the lightness, specialized the function so it supported only darkness values 8, 16, and 24 (which when rescaled to become 1, 2, and 3), and used a bitfield trick so that we had to perform only one expensive multiply instruction per iteration, rather than three, but that didn't seem to be helping. We'll now take advantage of the fact that the darkness factor is known to be 1, 2, or 3.

```
void darken(Pixel* first, Pixel* last, int darkness)
{
  int factor = darkness / 8;
  uint32_t mask2 = factor >= 2 ? 0xFFFFFFFF : 0;
  uint32_t mask3 = factor >= 3 ? 0xFFFFFFFF : 0;
  for (; first < last; ++first) {
    uint32_t v = first->v;
    uint32_t fields = (v & 0xFF) |
                      ((v & 0xFF00) << 2) |
                      ((v & 0xFF0000) << 4);
    fields += (fields & mask2) + (fields & mask3);
    fields += pack_fields(31, 31, 31);
    v -= (fields >> 5) & 0x1F;
    v -= (fields >> 7) & 0x1F00;
    v -= (fields >> 9) & 0x1F0000;
    first->v = v;
  }
}
```

The usage pattern for this function is that the darkness factor is in practice always 1, 2, or 3. So we calculate some masks that keep track of the actual value of the factor.

factor	`mask2`	`mask3`	`(fields & mask2)`	`+`	`(fields & mask3)`
1	`0x00000000`	`0x00000000`	`0`	`+`	`0`
2	`0xFFFFFFFF`	`0x00000000`	`fields`	`+`	`0`
3	`0xFFFFFFFF`	`0xFFFFFFFF`	`fields`	`+`	`fields`

The masks let us zero out one or both of the `fields` terms when calculating the product.

Alas, this is 2.2× slower than the previous version. It seems that performing two bitwise *and* operations and two additions is slower than a single multiply. My guess is that it's because the factor is so small, and the CPU has an early-out for small factors.

Okay, it's time to bring out the big guns. Time for the SIMD registers. We'll do that next time.

Raymond Chen

Follow