Optimizing code to darken a bitmap, part 2

devblogs.microsoft.com/oldnewthing/20220308-00

March 8, 2022

Raymond Chen

Our investigation into a simple function to darken a bitmap left off with this function:

```
struct 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);
  }
}
```

There's a lot of multiplication going on here, and multiplication tends to be one of the more expensive CPU instructions. Maybe we can collapse them into a single multiplication operation by running the calculations in parallel.

The idea here is to break the 32-bit integer into the respective channels, but spread them out so they act like independent lanes of a SIMD register. (This technique goes by the name SWAR, short for "SIMD within a register".) Once they've been spread out into lanes, we perform a single multiplication, which acts like a parallel multiplication across all the lanes, thanks to the magic of the distributive law:

$$(100^2a + 100b + 100c)d = 100^2ad + 100bd + cd$$

If $cd < 100$, then there will be no carry into the hundreds place, and similar, if $bd < 100$, then there will be no carry into the ten thousands place. Under such conditions, you can pluck out the individual products by extracting the corresponding pairs of digits from the final product.

It so happens that my program uses only three darkness values: 8, 16, and 24, corresponding to lightness factors 248, 240, and 232, respectively. The application of the lightness factor can therefore be simplified to

```
newPixel = oldPixel * (lightness / 8) / 32;
```

which can be rewritten as

```
newPixel = oldPixel - ceil(oldPixel * (darkness / 8) / 32);
```

The truncation toward zero in the lightness calculation becomes a ceiling calculation when viewed as darkness.

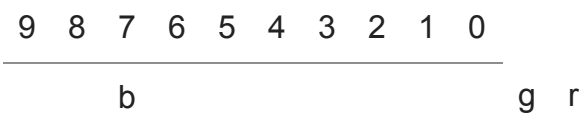
Reinterpreting it as a darkness calculation is helpful because the value of `darkness / 8` is limited to the values 1, 2, and 3. The factor is at most two bits.

Multiplying an 8-bit value with a 2-bit value produces a 10-bit result. And we can squeeze three 10-bit fields inside a 32-bit integer.

A quick mental check confirms that rounding up any fractional portion won't take us into 11 bits:

$255 \times 3 < 256 \times 3 = 256 \times 4 - 256 < 256 \times 4 - 32 = 10^{11} - 32$.

The idea therefore is that we take our three channel values and arrange them inside a 32-bit integer like this:



In other words,

- `value[ 0: 9] = r;`
- `value[10:19] = g;`
- `value[20:29] = b;`
- `value[30:31]` are unused

The lanes are 10 bits wide, so multiplying the entire integer by 1, 2, or 3 will not trigger any carries across lanes.

After the multiply, we reinterpret the integer as a fixed-point value, by applying an implied scale of $32 = 2^5$. Rounding up the fixed-point value to the next integer is the same as rounding up the original integer to the next multiple of 32.

Once the multiplication and rounding is done, we extract the integer portion of the fixed-point fields (which means we take the top five bits of each field) and subtract them from our starting value.

```
constexpr unsigned pack_fields(uint8_t r, uint8_t g, uint8_t b)
{
  return r | (g << 10) | (b << 20);
}

void darken(Pixel* first, Pixel* last, int darkness)
{
  int factor = darkness / 8;
  for (; first < last; ++first) {
    uint32_t fields = pack_fields(
        first->c[0], first->c[1], first->c[2]);
    fields *= factor;
    fields += pack_fields(31, 31, 31);
    first->c[0] -= (fields >>  5) & 31;
    first->c[1] -= (fields >> 15) & 31;
    first->c[2] -= (fields >> 25) & 31;
  }
}
```

Unfortunately, this benchmarks at 2.9× *slower* than the non-parallel version.

Okay, so maybe it's the byte access that is killing us. Let's load and store entire pixels at a time.

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

Still 2.1× slower than the non-parallel version. A tiny improvement but still way behind.

Some micro-tweaking here won't help much: We waste our time building up the difference fields, when we can just subtract each of the pieces as we calculate them.

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

As expected, this doesn't really help. Still 2.1× slower than the non-parallel version.

Part of the problem might be that I'm running the benchmarks on an x86-64 system, and the x86-64 does not have dedicated bitfield manipulation instructions, unlike other processors like PowerPC, ARM, and AArch64.

Or maybe the problem is that I'm doing a multiply. Since I know that the darkness factor is going to be 1, 2, or 3, I can strength-reduce that to an addition. We'll look at how to do that jumplessly next time.

Raymond Chen

Follow