Optimizing code to darken a bitmap, part 5

devblogs.microsoft.com/oldnewthing/20220311-00

March 11, 2022

Raymond Chen

For our last trick, we'll ARM-ify this simple function to darken a bitmap.

```
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);
  }
}
```

The general principle is the same, but we just apply it to ARM Neon intrinsics instead of x86 SSE intrinsics.

```
void darken(Pixel* first, Pixel* last, int darkness)
{
  int lightness = 256 - darkness;
  uint16x8_t lightness128 = vdupq_n_u16((uint16_t)lightness);
  lightness128 = vsetq_lane_u16(256, lightness128, 3);
  lightness128 = vsetq_lane_u16(256, lightness128, 7);
  void* end = last;
  for (auto pixels = (uint8_t*)first; pixels < end; pixels += 16) {
    uint8x16_t val = vld1q_u8(pixels);
    uint8x16x2_t zipped = vzipq_u8(val, vdupq_n_u8(0));
    uint16x8_t lo = vreinterpretq_u16_u8(zipped.val[0]);
    lo = vmulq_u16(lo, lightness128);
    auto hi = vreinterpretq_u16_u8(zipped.val[1]);
    hi = vmulq_u16(hi, lightness128);
    val = vuzpq_u8(vreinterpretq_u8_u16(lo), vreinterpretq_u8_u16(hi)).val[1];
    vst1q_u8(pixels, val);
  }
}
```

We want to set up our `lightness128` vector to consists of 8 lanes of 16-bit values, with the ones corresponding to color channels get the specified lightness, and the ones corresponding to alpha channels get a lightness of 256, which means "do not darken". The quickest way I found to do this (not that I looked very hard) is to broadcast the `lightness` value to all the lanes, and then set lanes 3 and 7 explicitly to 256.

Inside the loop, we process 16 bytes at a time, which comes out to four pixels.

First, we load the 16 bytes into a Neon register and call it `val`.

Next, we *zip* the 16-byte register with a register with a register full of zeroes. The zip intrinsic interleaves the results twice: The element in `val[0]` contains the interleaved low bytes, and the element in `val[1]` contains the interleaved high bytes.

source 0	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3
source 1	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3

`val[0]`	B7	A7	B6	A6	B5	A5	B4	A4	B3	A3	B2	A2	B1
`val[1]`	B15	A15	B14	A14	B13	A13	B12	A12	B11	A11	B10	A10	B9

If you aren't interested in one of the results, you can just ignore it, and the optimizer will remove that calculation from the code generation. In our case, we use both parts, though, so that optimization doesn't come into play.

Zipping with zero has the effect of zero-extending all the lanes from 8-bit to 16-bit, once you reinterpret the values as eight 16-bit lanes rather than sixteen 8-bit lanes.

For the low part, we take the first zipped-up value, reinterpret it as eight 16-bit lanes, and then perform a parallel multiply with our `lightness128` vector. Our x86 version took the result of the multiplication and shifted it right by 8 positions at this point, in order to divide by 256 and put the values in the right place to be combined with a pack instruction. For Neon, however, we'll leave the values in the odd bytes for reasons we'll see later.

We perform the same set of calculations on the high part, again leaving the values in the odd bytes of the result.

The final step is combining the results, which we do with the *unzip* instruction. This takes the even-numbered lanes from the first source register and puts them in the low-order lanes of the first destination. And it takes the even-numbered lanes from the second source register and puts them in the high-order lanes of the first destination. The odd-numbered lanes are collected and placed in the second destination. If you just relabel the bytes, you'll see that this is the inverse of the *zip* instruction, hence its name.

source 0	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3
source 1	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3
↓zip↓													
`val[0]`	B7	A7	B6	A6	B5	A5	B4	A4	B3	A3	B2	A2	B1
`val[1]`	B15	A15	B14	A14	B13	A13	B12	A12	B11	A11	B10	A10	B9

It is that second result that's interesting to us: For each 16-bit lane consists of An in the low byte and Bn in the high byte, we want to shift right 8 and save the low byte. "Shift right 8 and save the low byte" is the same as "extract the high byte", which in our case is Bn. In other words, we want to save all the Bn bytes and pack them into a single register. Everything is set up perfectly for an unzip, and our result is in the unzip output `val[1]`, which we store to memory.

I don't have easy access to an AArch64 system for performance testing,[1] so I can't say how much faster this is than the original version, but I suspect it gives a comparable speed-up as the x86 version.

[1] So how do I know this code even works? I put this function into a test program and sent it to a colleague who does have an AArch64 system. He ran the program and sent me a screen shot of the output, and I visually confirmed that it looked correct. It took me back to the days of punch card programming, where you submitted your deck to the machine operator and

then came back later for the results. Faced with such slow turnaround times, you double- and sometimes triple-checked that your program was exactly what you wanted, because if you messed up, you had to go through the whole cycle again.

Raymond Chen

Follow