Optimizing code to darken a bitmap, part 4

devblogs.microsoft.com/oldnewthing/20220310-00

March 10, 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 tried parallelizing the multiplication by treating a traditional 32-register as a bunch of 10-bit fields, but it turns out that all the overhead of shifting and masking ended up costing more than the savings of reducing the number of multiplications.

But instead of doing fake SIMD, let’s just do real SIMD.

First, we’ll use the x86 SIMD intrinsics. I’m going to limit myself to SSE2, since that’s the minimum requirement for x86-based Windows starting with Windows 8.

For simplicity of exposition, let’s assume that the start of the pixel array is aligned on a 16-byte boundary and the total size is a perfect multiple of 16. This avoids the hassle of dealing with the edge cases at the start and end.

```
void darken(Pixel* first, Pixel* last, int darkness)
{
  int lightness = 256 - darkness;
  auto lightness128 = _mm_set_epi16(
        256, lightness, lightness, lightness,
        256, lightness, lightness, lightness);
  void* end = last;
  for (auto pixels = (__m128i*)first; pixels < end; pixels++) {
    auto val = _mm_loadu_si128(pixels);
    auto vlo = _mm_unpacklo_epi8(val, _mm_setzero_si128());
    vlo = _mm_mullo_epi16(vlo, alpha128);
    vlo = _mm_srli_epi16(vlo, 8);
    auto vhi = _mm_unpackhi_epi8(val, _mm_setzero_si128());
    vhi = _mm_mullo_epi16(vhi, alpha128);
    vhi = _mm_srli_epi16(vhi, 8);
    val = _mm_packus_epi16(vlo, vhi);
    _mm_storeu_si128(pixels, val);
  }
}
```

First, we set up our `lightness128` vector to consists of eight 16-bit lanes. The lanes corresponding to color channels get the specified lightness, and the lanes corresponding to alpha channels get a lightness of 256, which means "do not darken".

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

First, we load the 16 bytes into an SSE2 register and call it `val`.

Next, we unpack the low part of the register with a register full of zeroes, putting the result into `vlo`. The "unpack low" instruction interleaves the low bytes of the two source registers.

source 1		A7		A6		A5		A4		A3		A2
source 2	B7	↓	B6	↓	B5	↓	B4	↓	B3	↓	B2	↓
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
destination	D15	D14	D13	D12	D11	D10	D09	D08	D07	D06	D05	D04

In our case, the second source register is all zeroes, so it has the effect of performing a zero extension of the first eight 8-bit values (corresponding to the first two pixels) into eight 16-bit values.

```
    auto vlo = _mm_unpacklo_epi8(val, _mm_setzero_si128());
```

source 1	A7	A6	A5	A4	A3	A2	A1	A0

source 2	00	↓	00	↓	00	↓	00	↓	00	↓	00	↓	00	↓	00	↓
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
destination	00	A7	00	A6	00	A5	00	A4	00	A3	00	A2	00	A1	00	A0
	A7		A6		A5		A4		A3		A2		A1		A0	

Next up is the multiplication and division:

```
vlo = _mm_mullo_epi16(vlo, alpha128);
vlo = _mm_srli_epi16(vlo, 8);
```

We perform a parallel multiply of the 16-bit values against the values in our `lightness128` register, and then we perform a parallel right-shift by 8 positions.

This combination of operations performs the `newPixel = oldPixel * lightness / 256` calculation on eight values at once. Recall that we preloaded the alpha channel with a lightness value of 256, so this multiplies by 256 and the shifts right by 8, which is a net nop.

We perform the same sequence of operations on the high bytes. The only difference is that we unpack with the `unpackhi` flavor of the intrinsic, so that it operates on the high 8 bytes instead of the low 8 bytes, thereby performing the calculations on the last two pixels instead of the first two.

We now have the desired results in sixteen 16-bit lanes, spread over two registers. We want to collapse those 16-bit lanes back into sixteen 8-bit lanes of a single register, which we do with the `pack` instruction. The `us` suffix means that this uses unsigned saturation. The unsigned part is important, but the saturation part isn't, since we know that the values are already in the range 0…255.

source 1	A7		A6		A5		A4		A3		A2		A1		A0
		↓		↓		↓		↓		↓		↓		↓	
destination	B7	A7	B6	A6	B5	A5	B4	A4	B3	A3	B2	A2	B1	A1	B0
	↑		↑		↑		↑		↑		↑		↑		↑
source 2	B7	B6		B5		B4		B3		B2		B1		B0	

At each iteration of the loop, we process four pixels.

This rewrite of the loop using SIMD pays off: It's 3.5 times faster then the non-SIMD version.

Next time, we'll apply the same approach to the ARM version.

Bonus chatter: I tried reducing the strength of the multiplication by using the same "addition with masking" trick that I tried in the general-purpose register version. It didn't help. The multiplication is fast enough that attempts to reduce its strength end up costing more in overhead than they do in savings by avoiding the multiplcation instruction.

Raymond Chen

Follow