SSE

Motivation

- Sometimes, SIMD doesn't quite provide what we want
- We need to synthesize the desired operation out of the building blocks we *do* have.

NOT

- Suppose we have image data, as RGBA
- Invert all colors of an image: There’s no bitwise NOT operation
- How can we accomplish this?

XOR

- We can press XOR into service

```
__m256* p = (__m256i*) img.pixels();
__m256i ones = _mm256_cmpeq_epi8(ones,ones);
for(unsigned i=0;i<n;i+=32,p++){
    v = _mm256_lddqu_si256(p);
    v = _mm256_xor_si256(v,ones);
    _mm256_storeu_si256(p,v);
}
```

- But there's a problem. What is it?

Problem

- This also inverts the *alpha* channel
  - That converts image to be entirely transparent.
  - We want to just invert the RGB channels.
- How to do this?

Solution

- Change our mask:

```
__m256* p = (__m256i*) img.pixels();
__m256i ones = _mm256_set_epi8( 255,255,255,0, 255,255,255,0,
    255,255,255,0, ... );
for(unsigned i=0;i<n;i+=32,p++){
    v = _mm256_lddqu_si256(p);
    v = _mm256_xor_si256(v,ones);
    _mm256_storeu_si256(p,v);
}
```

Averaging

- Suppose we want to convert image to greyscale
- Several ways to accomplish this
- Option 1: Replicate one channel to all slots
  - Human eye most sensitive to green, so that’s a logical choice
- This is just a swizzle operation

Code

```
uint8_t tmp[32] = {
        1,1,1,3,    5,5,5,7,  9,9,9,11,   13,13,13,15,
        1,1,1,3,    5,5,5,7,  9,9,9,11,   13,13,13,15,
    };
__m256i shuf = _mm256_lddqu_si256( (__m256i*) tmp);
__m256i* p = (__m256i*) img.pixels();
for(unsigned i=0;i<n;i+=32,p++){
    __m256i v = _mm256_load_si256(p);
    v = _mm256_shuffle_epi8(v, shuf );
    _mm256_storeu_si256(p,v);
}
```

- Timings: SIMD: 6009 μsec; non-SIMD: 20967 μsec

Problem

- What if we have r=255, g=0, b=255 (bright magenta)?
  - This scheme will output black

Strategy 2

- What if we want to take average?
- Compute: (r+g+b)/3
- Or: Could compute: r/3 + g/3 + b/3
- Which approach should we use?

Ex

- Suppose r=2, g=2, b=2.
- Then r/3+g/3+b/3 = 0+0+0 = 0
- What about (r+g+b)/3?
  - (2+2+2)/3 = 6/3 = 2
- So it looks like we want (r+g+b)/3
- But...We have a problem. What is it?

Problem

- Suppose r=255, g=255, b=255
- If we add with saturation: We get (255+255+255) = 255
  - Then 255/3 = 85
- If add without saturation: 255+255+255 = 253
  - 255+255=510 $\rightarrow$ 254
  - Then 254+255=509 $\rightarrow$ 253
  - And we get 253/3 = 84
- How to solve this problem?

Solution?

- Promote bytes to int, then divide by 3
- Unfortunately, there's no SSE integer divide instruction
  - Only floating point divide
- Darn.

Method

- Maybe we can use another method: *Lightness*
- Take max of r,g,b and min of r,g,b
- Sum those
- Divide by two
- But...Didn't we just decide there was no division operator?

Division

- Yes, but there's an instruction that applies concept of *strength reduction*
  - Division by 2 == Shift right one place
- _mm256_avg_epu8(a,b): Compute (a+b)>>1

Step 1

- Load eight pixels to a register

```
__m256i v1 = _mm256_lddqu_si256(p);
```

$$v1 = \begin{array}{c} (hi)\alpha_7, b_7, g_7, r_7, \alpha_6, b_6, g_6, r_6, \alpha_5, b_5, g_5, r_5, \alpha_4, b_4, g_4, r_4| \\ \alpha_3, b_3, g_3, r_3, \alpha_2, b_2, g_2, r_2, \alpha_1, b_1, g_1, r_1, \alpha_0, b_0, g_0, r_0(low) \end{array}$$

- First byte in memory (red, pixel 0) goes to slot 0 of YMM; last byte in memory (alpha, pixel 7) goes to slot 31 of YMM
- The | represents the lane division

Step 2

- Do a 32-bit right shifts on v1:
  __m256i v2 = _mm256_srli_epi32(v1, 8);
  __m256i v3 = _mm256_srli_epi32(v1, 16);

$$v2 = \begin{array}{c} 0, \alpha_7, b_7, g_7, 0, \alpha_6, b_6, g_6, 0, \alpha_5, b_5, g_5, 0, \alpha_4, b_4, g_4 | \\ 0, \alpha_3, b_3, g_3, 0, \alpha_2, b_2, g_2, 0, \alpha_1, b_1, g_1, 0, \alpha_0, b_0, g_0 \end{array}$$

$$v3 = \begin{array}{c} 0, 0, \alpha_7, b_7, 0, 0, \alpha_6, b_6, 0, 0, \alpha_5, b_5, 0, 0, \alpha_4, b_4 | \\ 0, 0, \alpha_3, b_3, 0, 0, \alpha_2, b_2, 0, 0, \alpha_1, b_1, 0, 0, \alpha_0, b_0 \end{array}$$

Step 3

- Take the max's and mins

```
max = _mm256_max_epu8(v1,v2);
max = _mm256_max_epu8(max,v3);
min = _mm256_min_epu8(v1,v2);
min = _mm256_min_epu8(min,v3);
```

- slots 0, 4, 8, 12, ... , 28 of "max" have maximums of (r,g,b) for the eight pixels
- slots 0, 4, 8, 12, ... , 28 of "min" have minimums of (r,g,b) for the eight pixels
- slots 3, 7, 11, ..., 31 of "max" have original α values (since we always max'd with 0)

Step 4

- Take the average:
  avg = _mm256_avg_epu8(max,min);

Step 5

- Mask the parts where α will go:
- Let mask = {255 (byte 0),255,255,0,255,255,255,0,...,255,255,255,0 (byte 31)}
  - avg = _mm256_and_si256(avg, mask)
- Mask the α from the original
  - alpha = _mm256_andnot_si256(mask, v1);
- Combine:
  - avg = _mm256_or_si256(avg, alpha);

Shuffle

- Now we can distribute the values back to their respective pixel locations
- Let shuf = {0,0,0,3, 4,4,4,7, 8,8,8,11, 12,12,12,15, 0,0,0,3, 4,4,4,7, 8,8,8,11, 12,12,12,15};
  - Byte 0 of shuf=0; byte 31 of shuf=15
- Then:
  res = _mm256_shuffle_epi8(avg, shuf);

Timing

- For 3648x2736 image, -O3:
  - SIMD: 5617 μsec
  - Non-SIMD: 21612 μsec

Option

- Yet another option: Take account of qualities of human perception
  - Eye is more sensitive to green than red and more sensitive to red than blue
- NTSC method: $0.21r + 0.72g + 0.07b$

Option

- To avoid floating point conversions, we can alter this a bit:
  - $\frac{1}{4} \cdot r \rightarrow 0.25$
  - $\frac{11}{16} \cdot g \rightarrow 0.6875$
  - $\frac{1}{16} \cdot b \rightarrow 0.0625$
- Divide by 4: Shift right 2
- Divide by 16: Shift right by 4
- But what about 11/16?

Fraction

- Could divide by 16 then multiply by 11
  - But: This loses precision
- Ex: Suppose green = 111
  - Correct result: 11/16*100 = 76.3125 = 76
  - If we divide by 16, we get 6; multiply by 11 gives 66
  - Error: 13%
- Could multiply by 11 and then divide by 16
  - But: This overflows whenever green > 23
- Ideas...?

Idea

- One way:
  - Observe: $\frac{11}{16} = \frac{8+2+1}{16} = \frac{8}{16} + \frac{2}{16} + \frac{1}{16} = \frac{1}{2} + \frac{1}{8} + \frac{1}{16}$
  - So we could compute g/2 + g/8 + g/16
  - Thus:
    - tmp1 = g>>1
    - tmp2 = g>>3
    - tmp3 = g>>4
    - Result = tmp1+tmp2+tmp3
  - Ex: For 111: We get 55+13+8 = 74 (error = 2.6%)

Note

- If we wanted to be more exact, we could use
  - $\frac{7}{32} \cdot r = 0.21875 \cdot r$ (compared to 0.21*red for NTSC)
  - $\frac{23}{32} \cdot g = 0.7185 \cdot g$ (compared to 0.72*green)
  - $\frac{2}{32} \cdot b = 0.0625 \cdot b$ (compared to 0.07*blue)

Note

- This would be a bit more work:
  - $\left(\frac{4}{32}+\frac{2}{32}+\frac{1}{32}\right)r=\left(\frac{1}{8}+\frac{1}{16}+\frac{1}{32}\right)r$ = (r >> 3) + (r >> 4) + (r >> 5)
  - $\left(\frac{16}{32}+\frac{4}{32}+\frac{2}{32}+\frac{1}{32}\right)g=\left(\frac{1}{2}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}\right)g$ = (g>>1) + (g>>3) + (g >> 4) + (g >> 5)
  - $\left(\frac{2}{32}\right)b=\left(\frac{1}{16}\right)b$ = b >> 4
- We can choose whether we want to trade off speed for accuracy

Problem

- There's no 8-bit integer shift operation in AVX (or SSE, for that matter)
- There are 16, 32, and 64 bit integer shifts
- But we can build an 8-bit shift out of a larger shift + mask operation

Consider

- Suppose we have a YMM register with a series of bytes $b_0, b_1, \ldots$ and we want to shift each byte to the right by 3
- Register contents:
$$\begin{array}{c} b_{31,7}b_{31,6}b_{31,5}b_{31,4}b_{31,3}b_{31,2}b_{31,1}b_{31,0} \\ b_{30,7}b_{30,6}b_{30,5}b_{30,4}b_{30,3}b_{30,2}b_{30,1}b_{30,0} \\ \vdots \\ b_{0,7}b_{0,6}b_{0,5}b_{0,4}b_{0,3}b_{0,2}b_{0,1}b_{0,0} \end{array}$$
- Notation: $b_{i,j}$= bit j of byte i

Shift

- Suppose we do a 16-bit integer logical right shift: _mm256_srli_epi16(v, 3)
- Register contents before shift:

$$b_{31,7}b_{31,6}b_{31,5}b_{31,4}b_{31,3}b_{31,2}b_{31,1}b_{31,0}$$
$$b_{30,7}b_{30,6}b_{30,5}b_{30,4}b_{30,3}b_{30,2}b_{30,1}b_{30,0}$$
$$\vdots$$
$$b_{1,7}b_{1,6}b_{1,5}b_{1,4}b_{1,3}b_{1,2}b_{1,1}b_{1,0}$$
$$b_{0,7}b_{0,6}b_{0,5}b_{0,4}b_{0,3}b_{0,2}b_{0,1}b_{0,0}$$

Shift

- After shift:

$$\begin{gathered}000b_{31,7}b_{31,6}b_{31,5}b_{31,4}b_{31,3}\\ b_{31,2}b_{31,1}b_{31,0}b_{30,7}b_{30,6}b_{30,5}b_{30,4}b_{30,3}\\ \vdots\\ 000b_{1,7}b_{1,6}b_{1,5}b_{1,4}b_{1,3}\\ b_{1,2}b_{1,1}b_{1,0}b_{0,7}b_{0,6}b_{0,5}b_{0,4}b_{0,3}\end{gathered}$$

- Bytes 1,3,5,... are OK: They got zeros shifted in, like we wanted
- But byte 0 contains some “spillover” from byte 1
  - Likewise for bytes 2, 4, 6, ...

Mask

- Suppose we have a mask: 0b0001 1111 = 0x1f in all 32 positions
- We could then AND our result with that mask
- This would give us the result we want for an unsigned byte-shift by 3

Code

- We can write a function to help us out:

```
__m256i shiftRightByte(__m256i v, unsigned count){
    v = _mm256_srli_epi16( v, count);
    __m256i mask = _mm256_set1_epi8( 0xff >> count );
    return _mm256_and_si256( v, mask );
}
```

What We Need

- $\frac{1}{4} \cdot r \rightarrow$ Shift right 2
- $\frac{11}{16} \cdot g \rightarrow \frac{1}{2}g + \frac{1}{8}g + \frac{1}{16}g \rightarrow$ Shift 1, shift 3, shift 4
- $\frac{1}{16} \cdot b \rightarrow$ Shift right 4

Problem

- We seem to have another problem: We have different operations to apply to red, green, and blue channels
- SIMD is happiest when we can apply uniform operations across the whole register
- We'll deal with this next time!

Assignment

- None!
  - Just get caught up on the existing labs...

Sources

- https://software.intel.com/en-us/articles/intel-software-development-emulator#faq
- http://www.tomshardware.com/reviews/intel-drops-pentium-brand,1832-2.html
- https://www.mathworks.com/matlabcentral/answers/93455-what-is-the-sse2-instruction-set-how-can-i-check-to-see-if-my-processor-supports-it?requestedDomain=www.mathworks.com
- http://softpixel.com/cwright/programming/simd/ssse3.php
- https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf
- https://msdn.microsoft.com/en-us/library/y08s279d(v=vs.100).aspx
- http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing?page=0,2
- http://en.cppreference.com/w/cpp/language/alignas
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- MSDN SSE reference
- https://wiki.multimedia.cx/index.php/YUV4MPEG2
- http://ok-cleek.com/blogs/?p=20540
- http://www.liranuna.com/sse-intrinsics-optimizations-in-popular-compilers/
- https://www.cilkplus.org/tutorial-pragma-simd
- A. Ortiz. “Teaching the SIMD Execution Model: Assembling a Few Parallel Programming Skills.” Proceedings of 2003 ACM SIGCSE.
- Nasm documentation.
- Greggo. x86 - SSE Compare Packed Unsigned Bytes. https://stackoverflow.com/questions/16204663/sse-compare-packed-unsigned-bytes
- John D. Cook. Converting color to grayscale. https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org