SSE

Motivation

- Looking at more find-and-replace operations using SSE/AVX

Find-and-Replace

- We want to scan a chunk of data and replace all instances of the byte value S with the value R
  - Also called “mask and merge”
- How can we do this?

Find-and-Replace

- We need some variables:
  - search: The byte we're searching for, repeated 32 times
  - repl: The replacement byte, replicated 32 times
    - _mm256_set1_epi8(v)
  - input: The input data (16 or 32 bytes)
- First, we must figure out where the search value is
- There are a few ways to perform this operation...

Search

- Option 1:
  - __m256i mask = _mm256_cmpeq_epi8(input,search)
    - This sets the bytes that have the search value to 0xff
    - Everywhere else is 0x00

Example

- Suppose search is {3,3,3,...}
- Suppose input is {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,...}
- __m256i mask = _mm256_cmpeq_epi8(input,search)
- Then mask= {255,0,0,0,0,0,0,0,0,255,0,0,0,0,0,255,0,255,...}

Search

- Option 2:
- We can also use the string functions we previously saw
- But these are limited to 16-byte chunks (unlike AVX)
  - `__m128i mask = _mm_cmpestrm( input, 16, search, 16, _SIDD_UBYTE_OPS | _SIDD_CMP_EQUAL_EACH | _SIDD_UNIT_MASK )`
- Same end result: 0xff where there was a match, 0x00 elsewhere

Masking

- Next step: Take input, zero out slots where there was a match
- tmp = _mm256_andnot_si256(mask,input)
  - Recall: andnot = ~mask & input
- Let mask' be shorthand for ~mask:
  - tmp[i] = 0 if mask'[i] was zero (i.e., if mask[i]==255 $\rightarrow$ input[i] == search)
  - tmp[i] = input[i] if mask'[i] was 0xff (i.e., mask[i] == 0 $\rightarrow$ input[i] $\neq$ search)

Example

- search ={3,3,3,...}
- input = {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,...}
- mask = {255,0,0,0,0,0,0,0,0,255,0,0,0,0,0,255,0,255,...}
- mask' = {0,255,255,255,255,255,255,255,255,0,255,255,255,255,255,0,255,0,...}
- tmp = {0,1,4,1,5,9,2,6,5,0,5,8,9,7,9,0,2,0,...}

Replacement

- Put replacement value in all slots where we had a match
- tmp2 = _mm256_and_si256(repl,mask)
- tmp2[i] = repl[i] if mask[i] == 255 (i.e., input[i] == search)
- tmp2[i] = 0 if mask[i] == 0 (i.e., input[i] $\neq$ search)

Example

- search ={3,3,3,...}
- input = {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,...}
- mask = {255,0,0,0,0,0,0,0,0,255,0,0,0,0,0,255,0,255,...}
- tmp = {0,1,4,1,5,9,2,6,5,0,5,8,9,7,9,0,2,0,...}
- repl = {42,42,42,42,42,...}
- tmp2 = {42,0,0,0,0,0,0,0,0,42,0,0,0,0,0,42,0,42,...}

Result

- Merge the two temporaries
- output = _mm256_or_si256(tmp,tmp2)
  - We could have used ’add’ instead

Example

- search ={3,3,3,...}
- input = {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,...}
- mask = {255,0,0,0,0,0,0,0,0,255,0,0,0,0,0,255,0,255,...}
- tmp = {0,1,4,1,5,9,2,6,5,0,5,8,9,7,9,0,2,0,...}
- repl = {42,42,42,42,42,...}
- tmp2 = {42,0,0,0,0,0,0,0,0,42,0,0,0,0,0,42,0,42,...}
- output = {42,1,4,1,5,9,2,6,5,42,5,8,9,7,9,42,2,42,...}

Summary

- So here's what we have:

```
...load search and repl...
for( ... ){
    ...load input...
    __m256i mask = _mm256_cmpeq_epi8(input,search);
    tmp = _mm256_andnot_si256(mask,input);
    tmp2 = _mm256_and_si256(repl,mask);
    output = _mm256_or_si256(tmp,tmp2);
    ...store output...
}
```

Alternate Approach

- We can also use the blend() intrinsic
- Recall:
  __m256i _mm256_blendv_epi8(a, b, mask)
  - For each of the 32 bytes:
    - If top bit of mask slot i == 0: Write a[i] to output[i]
    - Else, write b[i] to output[i]

Blend

- Do the comparison (cmpeq or string op) to set mask == 0xff in slots where we want to replace
- Then do
  output = _mm_blendv_epi8(input, repl, mask)
  - If mask[i] is zero, choose input[i] else choose repl[i]

Procedure

- In summary, we have:

```
...load search and repl...
for( ... ){
    ...load input...
    __m256i mask = _mm256_cmpeq_epi8(input,search);
    output = _mm_blendv_epi8( input, repl, mask );
    ...store output...
}
```

- Fewer instructions, but blend may have higher latency/lower throughput...
  - Need to benchmark to know which is faster

One More Thing...

- We can make our search-and-replace more flexible
- Ex: Suppose we want to replace any value *greater than 250* with 255
- Load search with {250,250,250,...,250}
- Load repl with {255,255,255,...255}
- Compare: Instead of using _mm256_cmpeq_epi8 [compare equal], use:
  mask = _mm256_cmpgt_epi8(input,search)
  - cmpgt = compare greater-than

Alternative

- If we're happy with doing 16-byte chunks at a time, we could use string intrinsics (range search):
  - Set search to [0,16,x,x,x,...]
    - x=Don't care
  - Then use range search:
    _mm_cmpestrm(search, 2, input, 16, _SIDD_UBYTE_OPS | _SIDD_CMP_RANGES | _SIDD_UNIT_MASK)
  - Now do blendv() as before

Note

- SSE (and AVX) instruction set has only two comparison operations for integer/short/byte values: Signed-greater-than and equal-to

```
#define signedGT(v1,v2)  _mm256_cmpgt_epi8(v1,v2)
#define signedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
```

- We'd like to have the full stable of operations ($>, <, \geq, \leq, =, \neq$)
- Let's tackle the easy ones first!
- How can we get signed-less-than?

<

- Just reverse order of operands to signed-greater-than

```
#define signedGT(v1,v2)  _mm256_cmpgt_epi8(v1,v2)
#define signedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define signedLT(v1,v2)  signedGT(v2,v1)
```

- How about signed-less-equal?

$\leq$

- Can get signed-less-equal by doing less and equal and or-ing:

```
#define signedGT(v1,v2)  _mm256_cmpgt_epi8(v1,v2)
#define signedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define signedLT(v1,v2)  signedGT(v2,v1)
#define signedLE(v1,v2)  _mm256_or_si256( signedLT(v1,v2),
   signedEQ(v1,v2) )
```

- What about signed greater-equal?

$\geq$

- Straightforward:

```
#define signedGT(v1,v2)  _mm256_cmpgt_epi8(v1,v2)
#define signedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define signedLT(v1,v2)  signedGT(v2,v1)
#define signedLE(v1,v2)  _mm256_or_si256( signedLT(v1,v2),
   signedEQ(v1,v2) )
#define signedGE(v1,v2)  _mm256_or_si256( signedGT(v1,v2),
   signedEQ(v1,v2) )
```

- How about signed-not-equal?
  - Hint: There's no bitwise "NOT" intrinsic!

$\neq$

- We just have to be a bit more inventive...
  - If $a \neq b$ then either $a > b$ or $a < b$

```
#define signedGT(v1,v2)  _mm256_cmpgt_epi8(v1,v2)
#define signedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define signedLT(v1,v2)  signedGT(v2,v1)
#define signedLE(v1,v2)  _mm256_or_si256( signedLT(v1,v2),
   signedEQ(v1,v2) )
#define signedGE(v1,v2)  _mm256_or_si256( signedGT(v1,v2),
   signedEQ(v1,v2) )
#define signedNE(v1,v2)  _mm256_or_si256( signedGT(v1,v2),
   signedLT(v1,v2) )
```

One More Thing...

- The previous scheme won't quite work in some cases
- Ex: If we're dealing with image data, we have *unsigned* values
- Ex: Suppose we want to replace any byte that has value > 250
  - Signed 8-bit integers can only represent -128...127
  - Any pixels with, say, blue of 255 will be treated as if blue is -128
  - This isn't what we want.

Note

- How to do *unsigned* comparison?
- Review: Representation of signed numbers...

One's Complement

- Most "obvious" way to represent numbers: One's complement (or sign-magnitude)
  - High bit represents sign; remainder represent value
- Consider very short (4 bit) integers
  - Same principle holds for 8 bit, 16 bit, 32 bit, 64 bit, ... integers
- We have exactly 16 possible bit combinations:

```
0000    0100    1000    1100
0001    0101    1001    1101
0010    0110    1010    1110
0011    0111    1011    1111
```

One's Complement

- In sign-magnitude, we use the top bit to hold the sign (0=+, 1=-), and the rest to hold the value (magnitude).

```
0000 = 0    1000 = -0
0001 = 1    1001 = -1
0010 = 2    1010 = -2
0011 = 3    1011 = -3
0100 = 4    1100 = -4
0101 = 5    1101 = -5
0110 = 6    1110 = -6
0111 = 7    1111 = -7
```

- But computers don't use this scheme

One's Complement

- One issue: There are *two* ways to represent zero
  - 00000000 (+0)
  - 10000000 (-0)
- Requires extra logic to compare +0 as equal to -0

One's Complement

- Another problem: Consider: 4+2
  - 0100 + 0010 = 0110 (6)
- But consider -4+-2. If we use the same logic as for positive addition:
  - 1100 + 1010 = 10110 $\rightarrow$ 0110 (6). Oops.
- Or -4+2:
  - 1100 + 0010 = 1110 (-6). Oops again.
- We need to create logic for four cases: (++), (+-), (-+), (- -)

Two's Complement

- Another way of representing negative numbers: *Two's complement*
- To negate a number: Invert all the bits and add 1 (discard any carry)

```
1000 = -8    1100 = -4    0000 = 0    0100 = 4
1001 = -7    1101 = -3    0001 = 1    0101 = 5
1010 = -6    1110 = -2    0010 = 2    0110 = 6
1011 = -5    1111 = -1    0011 = 3    0111 = 7
```

- Imagine we take 4+2: 0100 + 0010 = 0110 (6). Correct!
- Or if we have -4+2: 1100 + 0010 = 1110 = -2. Correct!
- Or -4+-2: 1100+1110 = 11010 = 1010 = -6. Correct!
- Or -1+1: 1111 + 0001 = 1 0000 = 0000 [discard 5th bit] = 0. Correct!

Problem

- Suppose a=1100, b=0011
- Is a<b?
  - Depends on whether a and b are signed or not!

Problem

- Suppose a=1100, b=0011
- Is a<b?
  - If unsigned: a=12, b=3 $\rightarrow$ False
  - If signed: a=-4, b=3 $\rightarrow$ True

SSE

- The greater-than operation only works for signed numbers
- But: We can use a trick to handle unsigned numbers:
  - If a == unsignedMax(a,b): Then a $\geq$ b in unsigned arithmetic
  - SSE/AVX *does* provide a max_epu8() function

$=, \geq$

- So we can start writing macros for our six unsigned comparison operations
- Equality is the same for both

```
#define unsignedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define unsignedGE(v1,v2)  _mm256_cmpeq_epi8( v1, _mm256_max_epu8(
   v1,v2) )
```

- How about the other operations?
- Which one is easiest?

$\leq$

- Can get less-equal by flipping operands

```
#define unsignedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define unsignedGE(v1,v2)  _mm256_cmpeq_epi8( v1, _mm256_max_epu8(
   v1,v2) )
#define unsignedLE(v1,v2)  unsignedGE(v2,v1)
```

- How about greater-than?

>

- How about a > b?
  - a is greater than b if:
    (a is not less than b) AND (a is not equal to b)
- Rewrite as:
  not (a < b) and not (a==b)
- Which can be rewritten:
  a≥b and not(a==b)
  - Remember, we have an and-not intrinsic: andnot(a,b) = ~a & b

$>$

```
#define unsignedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define unsignedGE(v1,v2)  _mm256_cmpeq_epi8( v1, _mm256_max_epu8(
   v1,v2) )
#define unsignedLE(v1,v2)  unsignedGE(v2,v1)
#define unsignedGT(v1,v2)  _mm256_andnot_si256( unsignedEQ(v1,v2),
    unsignedGE(v1,v2) )
```

- How about unsigned less-than?

<

- Easy peasy!

```
#define unsignedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define unsignedGE(v1,v2)  _mm256_cmpeq_epi8( v1, _mm256_max_epu8(
   v1,v2) )
#define unsignedLE(v1,v2)  unsignedGE(v2,v1)
#define unsignedGT(v1,v2)  _mm256_andnot_si256( unsignedEQ(v1,v2),
    unsignedGE(v1,v2) )
#define unsignedLT(v1,v2)  unsignedGT(v2,v1)
```

- Last one: $\neq$. How do we do this?

$\neq$

- Just as with signed $\neq$.

```
#define unsignedEQ(v1,v2)  _mm256_cmpeq_epi8(v1,v2)
#define unsignedGE(v1,v2)  _mm256_cmpeq_epi8( v1, _mm256_max_epu8(
   v1,v2) )
#define unsignedLE(v1,v2)  unsignedGE(v2,v1)
#define unsignedGT(v1,v2)  _mm256_andnot_si256( unsignedEQ(v1,v2),
    unsignedGE(v1,v2) )
#define unsignedLT(v1,v2)  unsignedGT(v2,v1)
#define unsignedNE(v1,v2)  _mm256_or_si256( unsignedLT(v1,v2) ,
   unsignedGT(v1,v2) )
```

Different Approach

- There is another way to determine if a>b for unsigned a,b
  - Let a' = a + INT_MIN
    - INT_MIN = -128 for 8 bit, -32768 for 16 bit, etc.
  - Let b' = b + INT_MIN
- Return result of signed comparison: a' > b'

Example

- Let a=125, b=129
  - a' = 125 + -128 = -3
  - b' = 129 + -128 = 1
  - Test: -3 > 1?
    - False
- We're basically sliding the numbers down the number line
  - Instead of being in range [0,255], they are in range [-128,127]

Third Way

- A third way: XOR with 0x80 (8 bit) or 0x8000 (16 bit) or 0x80000000 (32 bit)
- Then do ordinary signed compare
- Ex: Unsigned compare of 102 and 250
  - 102 = 0110 0110
  - 250 = 1111 1010
- XOR with 0x80 flips upper bit
  - 102 $\rightarrow$ 1110 0110 = -26 (signed)
  - 250 $\rightarrow$ 0111 1010 = 122 (signed)
- -26 < 122 (signed)

Assignment

- One interesting image transform is *posterization*
  - We reduce the number of colors in an image – sometimes quite significantly
  - This results in banding artifacts (which some people consider “artistic”)
- Write a program which takes a single command line argument: The filename of an image
- Posterize the image. Each channel should be treated independently:
  - $0 \ldots 63 \rightarrow 0$
  - $64 \ldots 127 \rightarrow 96$
  - $128 \ldots 191 \rightarrow 172$
  - $192 \ldots 255 \rightarrow 255$
- More follows...

Assignment

- Output to the file “posterized.png”
- Use AVX and process 32 bytes at a time. You can assume the inputs are in the format RGBA8.
  - The Image.h header always pads the image buffer so it has trailing padding when needed, so you won’t fall off the end of the buffer when you do load’s and store’s on an image that’s not a multiple of 32 bytes in size
- Example images are available on the class website; here are some files that you might find useful: nonsimd.cpp, Stopwatch.h, ymm.h
- Datapoint: On the “leaves.jpg” file, I got 50 msec without SIMD, 5 msec with SIMD

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