Swizzling

Motivation

- SSE intrinsics and datatypes
- We've seen arithmetic operations
- Now we'll look at *swizzling*

Swizzle

- Often, we need to adjust data layout before further processing
- SSE provides a number of data swizzle instructions
- But first, we need to examine details of data layout

Little Endian

- Intel CPU's are little endian
- Means LSB stored first in RAM
- Intel followed this convention for XMM registers too
  - First float in RAM goes to low slot of XMM register
- This might be inconvenient in some cases (ex: for matrices)
  - In that case use _mm_loadr_ps

Illustration

- Suppose we have this:
  alignas(16) float F[] = {1,2,3,4};
  __m128 v = _mm_load_ps(F);
- Question: Does the lowest slot of v get 1 or 4?
  - Does it matter?

Illustration

- We might not care!
- Consider:

```
__m128 v = _mm_load_ps(F);
__m128 w = _mm_set1_ps(42.0);
v = _mm_add_ps(v,w);
_mm_store_ps(F,v);
```

- Doesn't matter which end of v gets F[0]
  - Store will be consistent with load
  - That's all we need

Endian

- Let's take a look at an example instruction where it does make a difference
- Shuffle packed single precision
- __m128 v = _mm_shuffle_ps(__m128 a, __m128 b, uint8_t s)
  - s is 8 bits. Call the bits DDCCBBAA
  - v[0] = a[AA]
  - v[1] = a[BB]
  - v[2] = b[CC]
  - v[3] = b[DD]

Example

- Let's go back to our previous example and add a shuffle:

```
alignas(16) float F[] = {1,2,3,4};
__m128 v = _mm_load_ps(F);
//broadcast slot 0 to all outputs
__m128 x = _mm_shuffle_ps(v,v,0);
_mm_store_ps(F,x);
for(int i=0;i<4;++i)
    cout << F[i] << " ";
```

- This outputs: 1 1 1 1
- So the *first* thing in RAM goes to slot 0 of the XMM register

Shuffle

- Shuffle for doubles: v = _mm_shuffle_pd(__m128d a, __m128d b, uint8_t s)
  - If low bit of s is zero: Copy a[0] to v[0], else a[1] to v[0]
  - If second bit of s is zero: copy b[0] to v[1], else b[1] to v[1]
- Shuffle for ints: v = _mm_shuffle_epi32(__m128i a, __m128i b, int s)
  - Shuffles 32 bit chunks. Essentially the same as _mm_shuffle_ps

Shuffle

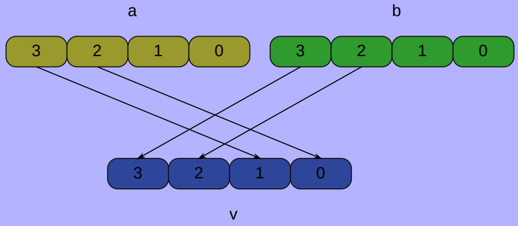
- Shuffle for bytes is more challenging because 16 bytes fit in one XMM register
- So an xmm argument is used instead
- __m128i v = _mm_shuffle_epi8(__m128i a, __m128i b)
  - b is treated as 16 single-byte elements
- For all 16 elements (i.e., let i=0...15):
  - If highest bit of b[i] is one: v[i] = 0
  - Else: v[i] = a[b[i] & 0xf]
- We can use this for shorts as well: Just set values of b to move adjacent elements around correctly

Note

- Handy function: Set xmm register to constant:
  __128i v = _mm_set_epi8(uint8_t slot15, uint8_t slot14, ... , uint8_t slot1, uint8_t slot0)
- Also set_epi16, set_epi32, set_epi64x
  - There's a set_epi64, but it uses __m64 type instead of __int64 type

Unpack

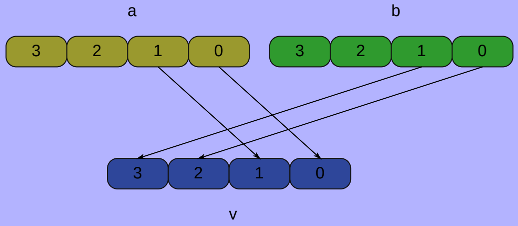
- Unpack: v = _mm_unpackhi_ps(a,b)
  - v[0] = a[2]
  - v[1] = b[2]
  - v[2] = a[3]
  - v[3] = b[3]



- More restricted version of shuffle

Unpack

- Unpack: v = _mm_unpacklo_ps(a,b)
  - v[0] = a[0]
  - v[1] = b[0]
  - v[2] = a[1]
  - v[3] = b[1]

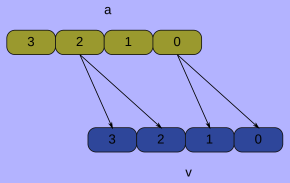


Unpack

- There's one for bytes: v = _mm_unpacklo_epi8(a,b)
  - v[0]=a[0]
  - v[1]=b[0]
  - v[2]=a[1]
  - v[3]=b[1]
  - v[4]=a[2]
  - ...
  - v[15]=b[7]
- And _mm_unpackhi_epi8(a,b)
  - Same idea, but uses a[8...15] and b[8...15]
- Similar instructions for 16, 32, 64 bit ints

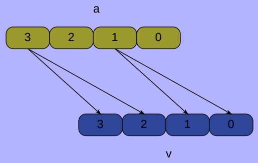
Replicate

- Replicate even items of XMM register
- __m128 v = _mm_moveldup_ps(a)
  - v[0] = a[0]
  - v[1] = a[0]
  - v[2] = a[2]
  - v[3] = a[2]



Replicate

- For odd items
- __m128 v = _mm_movehdup_ps(a)
  - v[0] = a[1]
  - v[1] = a[1]
  - v[2] = a[3]
  - v[3] = a[3]



Move

- Copy top two items of a to top two slots of result; copy top two items of b to bottom two slots of result
- __m128 v = _mm_movehl_ps(a, b)
  - v[0] = b[2]
  - v[1] = b[3]
  - v[2] = a[2]
  - v[3] = a[3]

Move

- Copy bottom two items of a to bottom two items of result; copy bottom two items of b to top two slots of result
- __m128 v = _mm_movelh_ps(a, b)
  - v[0] = a[0]
  - v[1] = a[1]
  - v[2] = b[0]
  - v[3] = b[1]

SSE4

- __m128 v = _mm_blend_ps(__m128 a, __m128 b, int s)
  - v[0] = (s & 1) ? b[0] : a[0]
  - v[1] = (s & 2) ? b[1] : a[1]
  - v[2] = (s & 4) ? b[2] : a[2]
  - v[3] = (s & 8) ? b[3] : a[3]
- Likewise for _mm_blend_epi16(__m128i a, __m128i b, int s)
  - 16 bit integers; uses low 8 bits of s
- And _mm_blend_epi32(__m128i a, __m128i b, int s)
  - Uses low four bits of s
- And _mm_blend_pd (for doubles))
  - Uses two bits from s
- Similar functions for AVX, but twice as many items in arguments

Blend

- For blending byte quantities, function is a bit different
- __m128i v = _mm_blendv_epi8(__m128i a, __m128i b, __m128i c)
  - v[i] = (c[i]) ? b[i] : a[i]
    - For i = 0...15
- __m256i v = _mm256_blendv_epi8(__m256i a, __m256i b, __m256i c)
  - Same thing but for 32 slots

Example

- Let’s see a real-world example of swizzling
- Suppose we want to do matrix-vector multiplication
- Suppose we have vector stored as __m128
- Our matrix is stored as four __m128’s (rows)
- Question: How to do multiplication?
  - (Work out in class)

Problem

- Not obvious how to do it
- The data we need is spread out among four __m128's
- Postmultiplying vector would be easy:
  - Dot products of matrix rows with vec4.
  - Done!

Problem

- If we stored matrix as *column major* it would be easy to do vector-matrix multiply
- But: C/C++ defaults to row-major order
  - Changing to column major might involve lots of data shuffling at load/store time
- And: We can't do matrix-vector multiply easily if we store data as column major

Swizzle

- This is where swizzling can help
- When we want to multiply:
  - Compute transpose of matrix
  - Then do dot product of vector with rows
- Computing transpose is a frequent operation, so we'd like to be able to do it anyway
- How to do?
  - (Work out in class)

Transpose

- Suppose we have four matrix rows in r1, r2, r3, r4
- Declare temporaries w,x,y,z
- Outputs: tr1,tr2,tr3,tr4: The transposed rows
- How to do?

r1	1,2,3,4
r2	5,6,7,8
r3	9,10,11,12
r4	13,14,15,16

w	
x	
y	
z	

tr1	
tr2	
tr3	
tr4	

Transpose

- w = _mm_unpackhi_ps(r1, r2);

r1	1,2,3,4
r2	5,6,7,8
r3	9,10,11,12
r4	13,14,15,16

w	3,7,4,8
x	
y	
z	

tr1	
tr2	
tr3	
tr4	

Test

- x = _mm_unpackhi_ps(r3, r4);

r1	1,2,3,4
r2	5,6,7,8
r3	9,10,11,12
r4	13,14,15,16

w	3,7,4,8
x	11,15,12,16
y	
z	

tr1	
tr2	
tr3	
tr4	

Transpose

- y = _mm_unpacklo_ps(r1, r2);

r1	1,2,3,4
r2	5,6,7,8
r3	9,10,11,12
r4	13,14,15,16

w	3,7,4,8
x	11,15,12,16
y	1,5,2,6
z	

tr1	
tr2	
tr3	
tr4	

Transpose

- z = _mm_unpacklo_ps(r3, r4);

r1	1,2,3,4		w	3,7,4,8		tr1	
r2	5,6,7,8		x	11,15,12,16		tr2	
r3	9,10,11,12		y	1,5,2,6		tr3	
r4	13,14,15,16		z	9,13,10,14		tr4	

Transpose

- tr1 = _mm_movelh_ps(y , z);

r1	1,2,3,4	w	3,7,4,8	tr1	1,5,9,13
r2	5,6,7,8	x	11,15,12,16	tr2	
r3	9,10,11,12	y	1,5,2,6	tr3	
r4	13,14,15,16	z	9,13,10,14	tr4	

Transpose

- tr2 = _mm_movehl_ps(z , y);

r1	1,2,3,4	w	3,7,4,8	tr1	1,5,9,13
r2	5,6,7,8	x	11,15,12,16	tr2	2,6,10,14
r3	9,10,11,12	y	1,5,2,6	tr3	
r4	13,14,15,16	z	9,13,10,14	tr4	

Transpose

- tr3 = _mm_movelh_ps(w , x);

r1	1,2,3,4		w	3,7,4,8		tr1	1,5,9,13
r2	5,6,7,8		x	11,15,12,16		tr2	2,6,10,14
r3	9,10,11,12		y	1,5,2,6		tr3	3,7,11,15
r4	13,14,15,16		z	9,13,10,14		tr4	

Transpose

- trow4 = _mm_movehl_ps(x , w);

r1	1,2,3,4	w	3,7,4,8	tr1	1,5,9,13
r2	5,6,7,8	x	11,15,12,16	tr2	2,6,10,14
r3	9,10,11,12	y	1,5,2,6	tr3	3,7,11,15
r4	13,14,15,16	z	9,13,10,14	tr4	4,8,12,16

Done!

- Now, tr1,tr2,tr3,tr4 are the transposed matrix's rows
- We can now use the dot product intrinsic
- Recall: _mm_dp_ps(a,b,f)
  - f = scalar: Flags
  - For vec4/vec4 dot product replicated to all four slots of output: f=0xff

Compute

- dpx = _mm_dp_ps(vec, tr1, 0xf1);
- dpy = _mm_dp_ps(vec, tr2, 0xf2);
- dpz = _mm_dp_ps(vec, tr3, 0xf4);
- dpw = _mm_dp_ps(vec, tr4, 0xf8);
- These are x,y,z,w of result

Store

- We need to combine these four items into one xmm register
- We can do this with bitwise-or
  - dpx = _mm_or_ps(dpx,dpy)
  - dpx = _mm_or_ps(dpx,dpz);
  - result = _mm_or_ps(dpx,dpw);
- Or, with blend:
  - tmp = _mm_blend_ps(dpx,dpy, 2);
  - tmp2 = _mm_blend_ps(dpz,dpw, 8);
  - result = _mm_blend_ps(tmp, tmp2, 0xc);

Assignment

- Write a program which takes a single command line argument. This will be the name of a two-track (stereo) wave file (some examples are found on the class webpage)
  - Exclusively using AVX intrinsics, swap the left and right channels
  - Write the output to a file named “swapped.wav”
  - For basic credit, support u8, s16, and f32 waves
  - For bonus [+50%], support s24 waves (again, using AVX intrinsics).
- Non-SSE program: chanswap.cpp

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
- http://ok-cleek.com/blogs/?p=20540
- http://www.liranuna.com/sse-intrinsics-optimizations-in-popular-compilers/
- https://www.cilkplus.org/tutorial-pragma-simd
- https://stackoverflow.com/questions/2804902/whats-the-difference-between-logical-sse-intrinsics
- Daniel Kusswurm. Modern X86 Assembly Language Programming. Apress.
- Peter Kankowski. Implementing strcmp, strlen, and strstr using SSE 4.2 instructions. https://www.strchr.com/strcmp_and_strlen_using_sse_4.2

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