SSE

Motivation

- Last time, we got an introduction to SSE
- Now we'll examine some more of its capabilities

Data Types

- We saw floats last time
- But SSE supports other types as well
- Double precision: Very similar to float
- Only difference is suffix on the intrinsics: pd instead of ps
- And how much data we can pack in a register

Loading

- Suppose dp is a pointer to double
- __m128d v = _mm_load_pd(dp)
- __m128d v = _mm_load1_pd(dp)
  - Replicates one double value to both xmm slots
- __m128d v = _mm_load_sd(dp)
  - Zeros upper half
- Remember: Alignment!

Storing

- _mm_store_pd(dp, v)
- _mm_store_pd1(dp, v)
- _mm_storer_ds(dp, v)
- _mm_storeu_pd(dp, v)

Arithmetic

- Again, just like float, but ‘pd’ suffix
  - No reciprocal or reciprocal square root though
- Add: v3 = _mm_add_pd(v1,v2)
- Subtract: v3 = _mm_sub_pd(v1,v2)
- Multiply: v3 = _mm_mul_pd(v1,v2)
- Divide: v3 = _mm_div_pd(v1,v2)
- Square root: v2 = _mm_sqrt_pd(v1)
- Minimum: v3 = _mm_min_pd(v1,v2)
- Maximum: v3 = _mm_max_pd(v1,v2)

AVX

- AVX works similarly, but registers are now 256 bits wide
- __m256 v = _mm256_load_ps(ptr)
  - Loads 8 floats to v
- __m256d v = _mm256_load_pd(ptr)
  - Loads 4 doubles to v

AVX Math

- AVX math is like SSE math
- __m256 v3 = _mm256_add_ps(v1,v2)
- Subtract: v3 = _mm256_sub_ps(v1,v2)
- Multiply: v3 = _mm256_mul_ps(v1,v2)
- Divide: v3 = _mm256_div_ps(v1,v2)
- Etc.

Horizontal

- Most SSE operations are “vertical”
- But there are some horizontal operations
- v3 = _mm_hadd_ps(v1,v2)
- v3[0] = v1[0] + v1[1]
- v3[1] = v1[2] + v1[3]
- v3[2] = v2[0] + v2[1]
- v3[3] = v2[2] + v2[3]
- Also hsub (subtraction)

Dot Product

- Since dot product is so common, Intel added an instruction just for that
- __m128 v = _mm_dp_ps(__m128 a, __m128 b, int s)
  - Let t = array of 4 floats (a temporary __m128)
    t[0] = (s&16 ? a[0]*b[0] : 0)
    t[1] = (s&32 ? a[1]*b[1] : 0)
    t[2] = (s&64 ? a[2]*b[2] : 0)
    t[3] = (s&128 ? a[3]*b[3] : 0)
    Let d = t[0]+t[1]+t[2]+t[3]
    v[0] = (s&1 ? d : 0)
    v[1] = (s&2 ? d : 0)
    v[2] = (s&4 ? d : 0)
    v[3] = (s&8 ? d : 0)

Usage

- Example: Compute dot product of two vec4's
  - v = _mm_dp_ps(a,b,0xff)
    - Replicate output to all slots
  - Or: v = _mm_dp_ps(a,b,0xf1)
    - Just set single output slot
- Example: Compute dot product of two vec3's
  - v = _mm_dp_ps(a,b,0x71)
- Example: Compute dot product of two vec2's
  - v = _mm_dp_ps(a,b,0x31)
- There's a _mm256_dp_ps intrinsic as well; it computes two dot products in one shot

Length

- We can use dot to create vector length function
- Suppose v is a vec4
- Compute squared length:
  dp = _mm_dp_ps(v,v,0xff)
- Take square root:
  len = _mm_sqrt_ps(dp)

Distance

- Hypotenuse function: Compute length of hypotenuse of triangle
- __m128 v = _mm_hypot_ps(a,b)
  - $v[0] = \sqrt{a[0]^2 + b[0]^2}$
  - $v[1] = \sqrt{a[1]^2 + b[1]^2}$
  - $v[2] = \sqrt{a[2]^2 + b[2]^2}$
  - $v[3] = \sqrt{a[3]^2 + b[3]^2}$
- How can we use this? Could we compute vector length with this?

vec2

- We can do some rearrangement of the data
- Suppose we have four vec2's: a, b, c, d
- Load data to __m128's:
- Let $X = [a_x, b_x, c_x, d_x]$
- Let $Y = [a_y, b_y, c_y, d_y]$
- Compute Q = _mm_hypot_ps(X,Y)
  - Q = $\left[\sqrt{a_x^2 + a_y^2}, \sqrt{b_x^2 + b_y^2}, \sqrt{c_x^2 + c_y^2}, \sqrt{d_x^2 + d_y^2}\right]$

vec4

- It's not easy to see how we can use hypot to speed up computation of any single length in 3D
- We make the SIMD more explicit
  - SIMD: Single Instruction Multiple Data
  - Perform same operation on several pieces of data
- Computing several lengths in parallel using primitive operations
- We'll start with vec4's

SIMD

- Suppose we have four vec4's: a,b,c,d
- We want the length of all four of them
- Let $X = [a_x, b_x, c_x, d_x]$
- Let $Y = [a_y, b_y, c_y, d_y]$
- Let $Z = [a_z, b_z, c_z, d_z]$
- Let $W = [a_w, b_w, c_w, d_w]$

Length

- Compute: Q = _mm_mul_ps(X,X)
  - Q = $[a_x^2, b_x^2, c_x^2, d_x^2]$
- Compute: R = _mm_mul_ps(Y,Y)
  - R = $[a_y^2, b_y^2, c_y^2, d_y^2]$
- Compute Q = _mm_add_ps(Q,R)
  - Q = $[a_x^2 + a_y^2, b_x^2 + b_y^2, c_x^2 + c_y^2, d_x^2 + d_y^2]$
- Compute: R = _mm_mul_ps(Z,Z)
  - R = $[a_z^2, b_z^2, c_z^2, d_z^2]$
- Compute: Q = _mm_add_ps(Q,R); R = _mm_mul_ps(W,W)
- Compute: Q = _mm_add_ps(Q, R); Q = _mm_sqrt_ps(Q)
  - Q = $\Big[\sqrt{a_x^2 + a_y^2 + a_z^2 + a_w^2}, \sqrt{b_x^2 + b_y^2 + b_z^2 + b_w^2}, \sqrt{c_x^2 + c_y^2 + c_z^2 + c_w^2}, \sqrt{d_x^2 + d_y^2 + d_z^2 + d_w^2}\Big]$

vec3

- How would we handle vec3's?

Integer Ops

- SSE also supports integer operations
- 16 byte register stores:
  - 16 bytes
  - 8 shorts
  - 4 ints
  - 2 long longs

Integer Ops

- int J[4] = ...;
- __m128i v = _mm_load_si128((__m128i*) J);
  - Pulls in 16 bytes of data
  - No distinction between ints, shorts, bytes, etc.
- _mm_store_si128(J,v)
- Alignment also important here

Arithmetic

- Addition: __m128i v3 = _mm_add_epiNN(v1, v2)
  - NN can be 8, 16, 32, or 64
- Subtraction: Same thing, but use 'sub'
- Multiply: _mm_mullo_epiNN(v1,v2)
  - NN can be 16 or 32
  - Multiplies items, discards any overflow, stores results
- No division operation available!
- Absolute value: _mm_abs_epiNN
  - NN can be 8, 16, or 32

Saturation

- We also have ‘saturation arithmetic’ operations
- _mm_adds_epXNN(v1,v2)
  - X is either i or u (signed or unsigned)
  - NN is either 8 or 16
  - Add with saturation
- _mm_subs_epXNN(v1,v2)
- Toy example: ex2.cpp
- Output:

  -6 -1 -2 0

Bitwise

- We have bitwise operations as well: v is of type __m128i
- _mm_slli_epi32(__m128i v, int count)
  - Shift left logical immediate
  - Also 16, 64 bit variants
  - Also srai and srli for shift right arithmetic and logical immediate

Bitwise

- _mm_sllv_epi32(__m128i v, __m128i count)
- Shift left logical variable
  - count[0...3] tells shift count for v[0...3]
  - Also 64 bit variant
  - Also srlv (shift right logical variable) and srav (shift right arithmetic variable)
- There are _mm256_ versions of these instructions too

Bitwise

- We have or, xor, and, andnot
- PFX is either mm or mm256
- NNN is either 128 or 256
  - _PFX_or_siNNN(v,w)
  - _PFX_xor_siNNN(v,w)
  - _PFX_and_siNNN(v,w)
  - _PFX_andnot_siNNN(v,w)
    - ~v & w

Conversion

- All of these take two arguments since we're shortening the data by half
- Convert eight ints (two arguments of four each) to eight shorts, with saturation
  - _mm_packs_epi32 $\rightarrow$ signed
  - _mm_packus_epi32 $\rightarrow$ unsigned
- Convert sixteen shorts (eight per argument) to sixteen bytes, with saturation
  - _mm_packs_epi16 $\rightarrow$ signed
  - _mm_packus_epi16 $\rightarrow$ unsigned

Extraction

- If we want to extract a specific slot, we can do so:
  - int v = _mm_extract_epi8(__m128i a, int b)
    - b = 0...15
    - Upper 3 bytes of v are zero
- Also have epi16, epi32, epi64

Gather

- AVX also introduced concept of *gather*: Can load noncontiguous elements
- __m128i _mm_i32gather_epi32(int* ptr, __m128i offset, int scale)
  - offset is treated as four int's
  - scale is 1, 2, 4, or 8
- Let R be the result. Operation:
  - Loads ptr+offset[0]*scale →R[0]
  - Loads ptr+offset[1]*scale →R[1]
  - Loads ptr+offset[2]*scale →R[2]
  - Loads ptr+offset[3]*scale →R[3]

Assignment

- Write a program which takes a two command line arguments.
  - The first will be the filename of an image file.
  - The second will be a number from -255 to 255.
  - Load the image file and increase its brightness by the specified amount.
  - Save the file to "out.png" in the current working directory.
  - Use SIMD to perform the operation.
- Report the time difference between the non-SIMD program and your SIMD program.
- Some example image files
- I suppose you might want the Image.h file too.

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

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