SSE

Motivation

- Often, we have repetitive operations to be executed on several data items
- We can speed things up by doing them in parallel
- Concept: SIMD: Single Instruction Multiple Data
- To take advantage of this, we might need to reorganize our code/data structures

Background

- We'll only consider x86 architecture here
  - ARM has similar instructions (Neon)
  - Other platforms have SIMD instructions too
- Basic idea:
  - Load several data items into a single register
  - Do one operation on all the data items in parallel
  - (Usually) store the data items back to RAM

Review

- Recall x86 CPU architecture...
  - Registers: Temporary storage locations
  - Used by CPU to hold data to be processed
  - 32 bit CPU: Seven GPR's: eax, ebx, ecx, edx, esi, edi, ebp
  - 64 bit CPU: Sixteen GPR's: rax, rbx, rcx, rdx, rsi, rdi, rbp, r8, r9, r10, r11, r12, r13, r14, r15

FP

- Floating point values are stored differently from integers
  - Like scientific notation
  - Ex: 3014.159 = 3.014159 × 103
- Since computers are binary, we use powers of 2, not powers of 10
  - Ex: $1.011011_2 \times 2^4$
- Float: 32 bits
  - 23 bits for mantissa, 8 bits for exponent, 1 bit for sign
- Double: 64 bits
  - 52 bits for mantissa, 11 bits for exponent, 1 bit for sign
- Extended: 80 bits
  - 62 bits for mantissa, 15 bits for exponent, 1 bit for sign, 1 bit for integer flag

FP

- x86 has special unit (FPU) to handle floating point operations
- Since FP numbers won't fit in 32 bit registers, Intel added 8 80-bit registers for FP values (way back in early 80's)
- Plus separate set of instructions to process them

Prehistory

- Early SIMD instruction set (1996): MMX
  - Officially, it doesn't stand for anything
    - You can't trademark an acronym!
  - But it could stand for "MultiMedia eXtensions"
- Used part of FP registers to operate on 64 bits at a time
  - CPU was only 32 bits, so couldn't use GPR's
- Limited, clunky, slow
- We ignore this

Stone Age

- Intel's next try: SSE (Pentium III, 1999)
- SSE stands for "Streaming SIMD"
  - Apparently, Intel gave up on the whole trademark thing
- A set of new instructions + 8 new registers
  - 128 bits wide
  - xmm0 - xmm7

Bronze Age

- SSE2: Pentium 4, 2001
- Additional SSE instructions
- Still uses xmm registers

Iron Age

- SSE3: Pentium 4, 2004
- More SSE instructions
- Same xmm registers

Medieval Period

- SSSE3: Core 2
- Additional SSE instructions
  - Super Streaming SIMD Extensions?
- Same xmm registers

Victorian Era

- SSE4: Intel Core Penryn/Nehalem, 2007
- Yet more SSE instructions
- Same xmm registers, but 64 bit code has 8 more of them (xmm8-xmm15)

Modern Age

- AVX: Intel Core, 2011
- New 256 bit registers (YMM0-YMM15)
    - Note: XMM## = Lower half of YMM##
- Plus some instructions for these registers
- And some three-operand instructions (finally!)

The Future

- AVX-512: Mid-2017
- Adds 512 bit ZMM registers
- Not widely available
  - I.e., I don’t have a machine that supports these
  - So we ignore them

Code

- We'll focus on SSE/AVX/AVX2 since support is fairly widespread
- And they are reasonably easy to work with
  - Especially compared to the floating point registers
  - MMX has some snags that make it more difficult to mix with other code

Data Types

- XMM registers are 128 bits = 16 bytes
- They can be treated as:
  - 16 single-byte quantities
  - 8 shorts
  - 4 ints
  - 4 floats
  - 2 longs
  - 2 doubles

Data Types

- YMM registers are 256 bits = 32 bytes
- They can be treated as:
  - 32 single-bytes
  - 16 shorts
  - 8 ints
  - 8 floats
  - 4 longs
  - 4 doubles

Intrinsics

- Too difficult to use assembly language directly
- Compilers provide alternatives: *intrinsics*
- They look like ordinary functions, but they translate directly into assembly instructions

Using

- GCC/Intel compilers:
  - #include <mmintrin.h> //mmx
  - #include <xmmintrin.h> //sse
  - #include <emmintrin.h> //sse2
  - #include <pmmintrin.h> //sse3
  - #include <tmmintrin.h> //ssse3
  - #include <smmintrin.h> //sse4.1
  - #include <nmmintrin.h> //sse4.2
  - #include <immintrin.h> //avx, avx2
- In Visual Studio, most are in intrin.h, but a few are in immintrin.h

Compile

- gcc
  - Compile with -mfpmath=sse -mmmx -msse -msse2 -msse3 -mssse3 -msse4.1 -msse4.2 -msse4 -mavx -mavx2
- Visual Studio
  - I suppose there’s some options somewhere for this...

Note

- See `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#` for the full reference
- Bookmark this and put a copy under your pillow
- You will *need* this site!

Data Types

- 128-bit data types: __m128, __m128i, __m128d
  - m128i = Integers
  - m128d = floating point
  - They're the same bit-wise, but system can optimize if it knows what type's expected to go in the register
- 256-bit data types: __m256, __m256i, __m256d

Functions

- Naming pattern: _mm_**operation**_**type**
  - operation is the operation being performed (add, subtract, etc.)
  - type tells what kind of data
    - ps = packed floats (4 at a time)
    - pd = packed doubles (2 at a time)
    - si128 = integer values (byte, short, int)

Examples

- We’ll start off with the float variants
- Four floats per xmm register

load_ps

- Load parallel (packed?) single precision
  - float* p = ...
  - __m128 v = _mm_load_ps(p)
    - v[0] = *p
    - v[1] = *(p+1)
    - v[2] = *(p+2)
    - v[3] = *(p+3)
- Important: Address pointed to by p must be on 16 byte boundary

Alignment

- Note: Some load instructions require data to be 16-byte aligned
- If it's not: Processor will issue a fault
- Ex: This may or may not work:
  ```
  float F[4] = { ... };
  __m128 v = _mm_load_ps(F);
  ```
- C++ 11 provides an alignas operator we can use:
  ```
  alignas(16) float F[4] = {...};
  __m128 v = _mm_load_ps(F);
  ```

Problem

- Dynamically allocated memory (malloc, new) is not guaranteed to be aligned
- Likewise for memory in vectors
- So this has a chance of crashing:

```
float* F = new float[n];
...fill it up...
__m128 v = _mm_load_ps(F);
```

- Likewise:

```
vector<float> F(n);
...fill it up...
__m128 v = _mm_load_ps(F.data());
```

Solution

- We can define our own type:
  ```
  struct alignas(16) XMMFloat{
      float v[4];
  };
  ```
- Then we can use new and vectors properly:
  ```
  XMMFloat* F = new XMMFloat[n];
  ...fill it up...
  __m128 v = _mm_load_ps((float*)F);
  ```
- Likewise:
  ```
  vector<XMMFloat> F(n);
  ...fill it up...
  __m128 v = _mm_load_ps((float*)F.data());
  ```

load_ps1

- Load and copy one single precision item
  - float* p = ...
  - __m128 v = _mm_load1_ps(p)
    - v[0] = *p
    - v[1] = *p
    - v[2] = *p
    - v[3] = *p
- Notice: It replicates the same thing to four places
- The documentation isn't clear if p must be aligned

load_ss

- Load scalar single precision: Loads one item and zeros the rest
  - float* p = ...
  - __m128 v = _mm_load_ss(p)
    - v[0] = *p
    - v[1] = 0
    - v[2] = 0
    - v[3] = 0
- No required alignment

loadr_ps

- Load reversed parallel (packed) single precision
- __m128 v = _mm_loadr_ps(p)
  - v[3] = *p
  - v[2] = *(p+1)
  - v[1] = *(p+2)
  - v[0] = *(p+3)
- Must be aligned

loadu_ps

- Load unaligned packed single precision
- __m128 v = _mm_loadu_ps(p)
  - v[0] = *p
  - v[1] = *(p+1)
  - v[2] = *(p+2)
  - v[3] = *(p+3)
- Same as load_ps, but p need not be on a 16-byte aligned address
- Older CPU’s had speed penalty for unaligned; modern ones don’t (for *most* accesses)

Store

- v = __m128 value; p=float pointer
- _mm_store_ps(p, v)
  - Stores contents of v to p[0]...p[3]
- _mm_store_ps1(p, v)
  - Replicates v[0] four times
- _mm_storer_ps(p, v)
  - Stores in reverse order
- _mm_storeu_ps(p, v)
  - Unaligned store (maybe slower)

Example

- Example code: ex0.cpp
- Output:

```
3 1 4 5
0 0 0 0
```

Computation

- OK, so now we know how to load and store data.
- What can we do with it?

Arithmetic

- Add 4 floats in parallel: __m128 v3 = _mm_add_ps(v1,v2)
- Subtract: v3 = _mm_sub_ps(v1,v2)
- Multiply: v3 = _mm_mul_ps(v1,v2)
- Divide: v3 = _mm_div_ps(v1,v2)
- GCC provides overloads for +, -, *, and / as well as unary negation
  - VS might do the same thing...

Example

- Example: ex1.cpp
- Output:

```
12 3 10 10
```

Arithmetic

- Reciprocal: __m128 v2 = _mm_rcp_ps(v1)
- Reciprocal square root: v2 = _mm_rsqrt_ps(v1)
- Square root: v2 = _mm_sqrt_ps(v1)
- Zero: v = _mm_setzero_ps()
  - Zero out the whole register
- Minimum: v3 = _mm_min_ps(v1,v2)
- Maximum: v3 = _mm_max_ps(v1,v2)
- Ceiling: v2 = _mm_ceil_ps(v1)
- Floor: v2 = _mm_floor_ps(v1)
- Round: v2 = _mm_round_ps(v1, mode)
  - Mode = _MM_FROUND_TO_NEAREST_INT or _MM_FROUND_TO_ZERO

Example

- Let’s see a real-life example: Scaling loudness in a WAV file
- Explain: WAV files: Header followed by data
- Data format: Several possible...

PCM

- PCM: Easiest to understand
- Format code = 1 (integer) or 3 (float)
- Integer format: 8, 16, 24, or 32 bits per sample
    - 8 bit = unsigned, 16/24/32 bit = signed
- Float format: 32 bits per sample, -1...1 range
- Directly measures amplitude of signal
- Usually one or two channels

ADPCM

- ADPCM: Provides compression
- Input: 16 bit PCM value
- Output: 4 bit quantized value: The delta between previous sample and this one
- As long as signal doesn't change too rapidly, ADPCM gives good reproduction at 25% of the bitrate

μ-law

- Quantizes 14 bit samples to 8 bit
  - Gives increased resolution in critical areas of waveform without requiring larger sample size
- For range from -8192...8191:
  - Break range into 18 intervals of different sizes
    - Closer interval is to zero, smaller it is
    - Further interval is from zero, larger it is
  - Divide each interval into 16 pieces
- For each 14 bit input sample:
  - Determine which interval/piece it falls in
  - Output 8 bit value: 4 bits for interval indentifier, 4 for piece identifier
- Result: Input parts near 0 get encoded with more accuracy; further away get less accuracy
- Works well for speech; used in analog telephones

Code

- We'll assume PCM/float format
- Header to load wave files: Wave.h
- Stopwatch for timing: Stopwatch.h
- Non-SSE program to adjust loudness: loudness.cpp
- SSE program to adjust: loudnessxmm.cpp

Results

- Input: WAV file, about 1 minute in length
- Machine: Core i7
- Average of five executions (microseconds):

Non-SIMD	SIMD
4439	2529

Assignment

- Write a program which takes three command line parameters.
  - The first is the name of a WAV file encoded using the floating point format.
  - The second is the echo decay (a number from 0 to 1).
  - The third is the echo delay.
  - Using these parameters, add echo using SIMD intrinsics.
  - Write the output to the file "out.wav".
  - Report the total time required for the operation.
- There are example files on the class website.

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
- https://en.cppreference.com/w/cpp/language/alignas
- http://www.ent.mrt.ac.lk/rds/index.php/curriculum-support/dsd-projects/30-batch-07-dsd-projects/57-wav-to-adpcm-converter

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