

# Lecture 15

## **Floating point**

# Announcements

## Today's lecture

- SSE – wrapping up
- Floating point

# Recapping from last time: how do we vectorize?

- Original code

```
double a[N], b[N], c[N];  
for (i=0; i<N; i++) {  
    a[i] = sqrt(b[i] / c[i]);  
}
```

- Identify vector operations, reduce loop bound

```
for (i = 0; i < N; i+=2)  
    a[i:i+1] = vec_sqrt(b[i:i+1] / c[i:i+1]);
```

- Introduce SSE intrinsics

[software.intel.com/sites/landingpage/IntrinsicsGuide/](http://software.intel.com/sites/landingpage/IntrinsicsGuide/)

- Speedup  
due to vectorization: x1.7

```
double *a, *b, *c;  
__m128d v1, v2, v3;  
for (i=0; i<N; i+=2) {  
    v1 = _mm_load_pd(&b[i]);  
    v2 = _mm_load_pd(&c[i]);  
    v3 = _mm_div_pd(v1, v2);  
    v3 = _mm_sqrt_pd(v3);  
    _mm_store_pd(&a[i], v3);  
}
```

# C++ intrinsics

- C++ functions and datatypes that map directly onto 1 or more machine instructions
- Supported by all major compilers
- The interface provides 128 bit data types and operations on those datatypes

- `_m128` (float)
- `_m128d` (double)

- Data movement and initialization

- `mm_load_pd` (aligned load)
- `mm_store_pd`
- `mm_loadu_pd` (unaligned load)

```
__m128d v1, v2, v3;  
for (i=0; i<N; i+=2) {  
    v1 = _mm_load_pd(&b[i]);  
    v2 = _mm_load_pd(&c[i]);  
    v3 = _mm_div_pd(v1, v2);  
    v3 = _mm_sqrt_pd(v3);  
    _mm_store_pd(&a[i], v3);  
}
```

- Data may need to be aligned
- See [software.intel.com/sites/landingpage/IntrinsicsGuide](https://software.intel.com/sites/landingpage/IntrinsicsGuide)

# Intrinsics

[software.intel.com/sites/landingpage/IntrinsicsGuide](http://software.intel.com/sites/landingpage/IntrinsicsGuide)

## Synopsis

```
__m128d _mm_load_pd(double const* mem_addr)
```

```
#include "emmintrin.h"
```

Instruction: **movapd** xmm, m128

CPUID Flags: SSE2

```
flags          : fpu vme de pse tsc msr pae
mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx lm constant_tsc arch_perfmon
pebs bts rep_good aperfmperf pni dtes64
monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr
pdcm dca lahf_lm dts tpr_shadow
```

```
double *a, *b, *c;
__m128d v1, v2, v3;
for (i=0; i<N; i+=2) {
    v1 = _mm_load_pd(&b[i]);
    v2 = _mm_load_pd(&c[i]);
    v3 = _mm_div_pd(v1, v2);
    v3 = _mm_sqrt_pd(v3);
    _mm_store_pd(&a[i], v3);
}
```

## Description

Load 128-bits (composed of 2 packed double-precision (64-bit) floating-point elements) from memory into **dst**. **mem\_addr** must be aligned on a 16-byte boundary or a general-protection exception may be generated.

# SSE2 Cheat sheet

| Data transfer                      | Arithmetic                    | Compare          |
|------------------------------------|-------------------------------|------------------|
| MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm | ADD{SS/PS/SD/PD} xmm, mem/xmm | CMP{SS/PS/SD/PD} |
|                                    | SUB{SS/PS/SD/PD} xmm, mem/xmm |                  |
| MOV {H/L} {PS/PD} xmm, mem/xmm     | MUL{SS/PS/SD/PD} xmm, mem/xmm |                  |
|                                    | DIV{SS/PS/SD/PD} xmm, mem/xmm |                  |
| (load and store)                   | SQRT{SS/PS/SD/PD} mem/xmm     |                  |
|                                    | MAX {SS/PS/SD/PD} mem/xmm     |                  |
|                                    | MIN{SS/PS/SD/PD} mem/xmm      |                  |

xmm: one operand is a 128-bit SSE2 register

mem/xmm: other operand is in memory or an SSE2 register

{SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: four 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register

{A} 128-bit operand is aligned in memory

{U} the 128-bit operand is unaligned in memory

{H} move the high half of the 128-bit operand

{L} move the low half of the 128-bit operand

Krste Asanovic & Randy H. Katz

# Today's lecture

- SSE – wrapping up
- Floating point

# What is floating point?

- A representation
  - $\pm 2.5732... \times 10^{22}$
  - NaN  $\infty$
  - Single, double, extended precision (80 bits)
- A set of operations
  - $+ = * / \sqrt{\text{rem}}$
  - Comparison  $< \leq = \neq \geq >$
  - Conversions between different formats, binary to decimal
  - Exception handling
- Language and library support

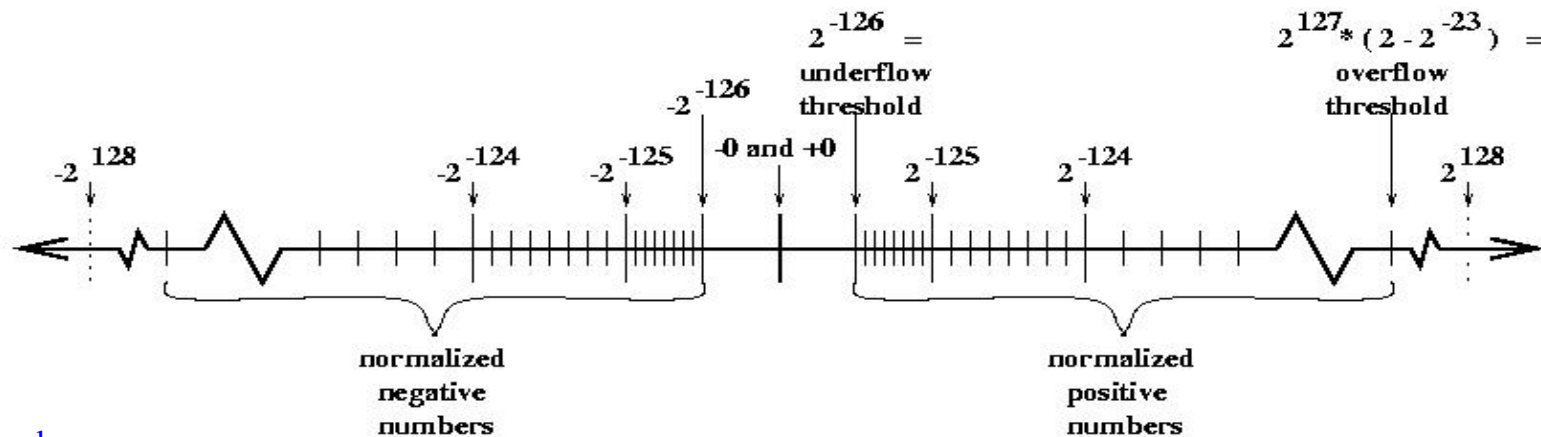
# IEEE Floating point standard P754

- Universally accepted standard for representing and using floating point, AKA “P754”
  - Universally accepted
  - W. Kahan received the Turing Award in 1989 for design of IEEE Floating Point Standard
  - Revision in 2008
- Introduces special values to represent infinities, signed zeroes, undefined values (“Not a Number”)
  - $1/0 = +\infty$  ,  $0/0 = \text{NaN}$
- Minimal standards for how numbers are represented
- Numerical exceptions

# Representation

- Normalized representation  $\pm 1.d\dots d \times 2^{\text{exp}}$ 
  - ▶ **Macheps** = Machine epsilon =  $\varepsilon = 2^{-\#\text{significant bits}}$   
relative error in each operation
  - ▶ **OV** = overflow threshold = largest number
  - ▶ **UN** = underflow threshold = smallest number
- $\pm$ Zero**:  $\pm$ significant and exponent = 0

| Format | # bits    | #significant bits | macheps                            | #exponent bits | exponent range                                  |
|--------|-----------|-------------------|------------------------------------|----------------|-------------------------------------------------|
| Single | 32        | 23+1              | $2^{-24}$ ( $\sim 10^{-7}$ )       | 8              | $2^{-126} - 2^{127}$ ( $\sim 10^{+-38}$ )       |
| Double | 64        | 52+1              | $2^{-53}$ ( $\sim 10^{-16}$ )      | 11             | $2^{-1022} - 2^{1023}$ ( $\sim 10^{+-308}$ )    |
| Double | $\geq 80$ | $\geq 64$         | $\leq 2^{-64}$ ( $\sim 10^{-19}$ ) | $\geq 15$      | $2^{-16382} - 2^{16383}$ ( $\sim 10^{+-4932}$ ) |



# What happens in a floating point operation?

- Round to the nearest representable floating point number that corresponds to the exact value (correct rounding)  
 $3.75 + .425 = .4175 \approx .418$  (3 significant digits, round toward even)
  - When there are ties, round to the nearest value with the lowest order bit = 0 (rounding toward nearest even)
- Applies to  $+$   $=$   $*$   $/$   $\sqrt{\text{rem}}$  and to format conversion
- Error formula:  $\text{fl}(a \text{ op } b) = (a \text{ op } b) * (1 + \delta)$  where
  - $\text{op}$  one of  $+$  ,  $-$  ,  $*$  ,  $/$
  - $|\delta| \leq \epsilon = \text{machine epsilon}$
  - assumes no overflow, underflow, or divide by zero
- Example

$$\begin{aligned}\text{fl}(x_1 + x_2 + x_3) &= [(x_1 + x_2) * (1 + \delta_1) + x_3] * (1 + \delta_2) \\ &= x_1 * (1 + \delta_1) * (1 + \delta_2) + x_2 * (1 + \delta_1) * (1 + \delta_2) + x_3 * (1 + \delta) \\ &\equiv x_1 * (1 + e_1) + x_2 * (1 + e_2) + x_3 * (1 + e_3) \\ &\quad \text{where } |e_i| \lesssim 2 * \text{machine epsilon}\end{aligned}$$

# Floating point numbers are not real numbers!

- Floating-point arithmetic does not satisfy the axioms of real arithmetic
- Trichotomy is not the only property of real arithmetic that does not hold for floats, nor even the most important
  - ▶ Addition is not associative
  - ▶ The distributive law does not hold
  - ▶ There are floating-point numbers without inverses
- It is not possible to specify a fixed-size arithmetic type that satisfies *all* of the properties of real arithmetic that we learned in school. The P754 committee decided to bend or break some of them, guided by some simple principles
  - ▶ When we can, we match the behavior of real arithmetic
  - ▶ When we can't, we try to make the violations as predictable and as easy to diagnose as possible (e.g. Underflow, infinities, etc.)

# Consequences

- Floating point arithmetic is not associative  
 $(x + y) + z \neq x + (y + z)$

- Example

$a = 1.0 \dots 0e+38$        $b = -1.0 \dots 0e+38$        $c = 1.0$

$(a+b)+c$ :  $1.0000000000000000e+00$

$a+(b+c)$ :  $0.0000000000000000e+00$

- When we add a list of numbers on multiple cores, we can get different answers
  - Can be confusing if we have a race conditions
  - Can even depend on the compiler
- Distributive law doesn't always hold
  - When  $y \approx z$ ,  $x*y - x*z \neq x(y-z)$
  - In Matlab v15, let  $x=1e38$ ,  $y=1.0+1.0e12$ ,  $z=1.0-1.0e-12$
  - $x*y - x*z = 2.0000e+26$ ,  $x*(y-z) = 2.0001e+26$

What is the most accurate way to sum the list of signed numbers  $(1e-10, 1e10, -1e10)$  when we have only 8 significant digits ?

- A. Just add the numbers in the given order
- B. Sort the numbers numerically, then add in sorted order
- C. Sort the numbers numerically, then pick off the largest of values coming from either end, repeating the process until done

# Exceptions

- An exception occurs when the result of a floating point operation is not a real number, or too extreme to represent accurately

$1/0$ ,  $\sqrt{-1}$   $1.0e-30 + 1.0e+30$   $1e38 * 1e38$

- P754 floating point exceptions aren't the same as C++11 exceptions
- The exception need not be disastrous (i.e. program failure)
  - ▶ Continue; tolerate the exception, repair the error

# An example

- Graph the function

$$f(x) = \sin(x) / x$$

- $f(0) = 1$
- But we get a singularity @  $x=0$ :  $1/x = \infty$
- This is an “accident” in how we represent the function (W. Kahan)
- We *catch* the exception (divide by 0)
- Substitute the value  $f(0) = 1$

Which of these expressions will generate an exception?

A.  $\sqrt{-1}$

B.  $0/0$

C.  $\text{Log}(-1)$

D. A and B

E. All of A, B, C

## Why is it important to handle exceptions properly?

- Crash of Air France flight #447 in the mid-atlantic  
<http://www.cs.berkeley.edu/~wkahan/15June12.pdf>
- Flight #447 encountered a violent thunderstorm at 35000 feet and super-cooled moisture clogged the probes measuring airspeed
- The autopilot couldn't handle the situation and relinquished control to the pilots
- It displayed the message **INVALID DATA** without explaining why
- Without knowing what was going wrong, the pilots were unable to correct the situation in time
- The aircraft stalled, crashing into the ocean 3 minutes later
- At 20,000 feet, the ice melted on the probes, but the pilots didn't know this so couldn't know which instruments to trust or distrust.

# Infinites

- $\pm\infty$
- Infinites extend the range of mathematical operators
  - $5 + \infty, 10 * \infty, \infty * \infty$
  - No exception: the result is exact
- How do we get infinity?
  - When the exact finite result is too large to represent accurately
  - Example:  $2 * OV$  [recall:  
OV = largest representable number]
  - We also get **Overflow exception**
- Divide by zero exception
  - Return  $\pm\infty = 1/\pm 0$

What is the value of the expression  $-1/(-\infty)$ ?

- A.  $-0$
- B.  $+0$
- C.  $\infty$
- D.  $-\infty$

## Signed zeroes

- We get a signed zero when the result is too small to be represented
- Example with 32 bit single precision

$$a = -1 / 1000000000000000000000000.0 == -0$$

$$b = 1 / a$$

- Because  $a$  is float, it will result in  $-\infty$  but the correct value is :  $-1000000000000000000000000.0$

| Format | # bits    | #significand bits | macheps                            | #exponent bits | exponent range                                  |
|--------|-----------|-------------------|------------------------------------|----------------|-------------------------------------------------|
| Single | 32        | 23+1              | $2^{-24}$ ( $\sim 10^{-7}$ )       | 8              | $2^{-126} - 2^{127}$ ( $\sim 10^{+-38}$ )       |
| Double | 64        | 52+1              | $2^{-53}$ ( $\sim 10^{-16}$ )      | 11             | $2^{-1022} - 2^{1023}$ ( $\sim 10^{+-308}$ )    |
| Double | $\geq 80$ | $\geq 64$         | $\leq 2^{-64}$ ( $\sim 10^{-19}$ ) | $\geq 15$      | $2^{-16382} - 2^{16383}$ ( $\sim 10^{+-4932}$ ) |

If we don't have signed zeroes, for which value(s) of  $x$  will the following equality not hold true:  $1/(1/x) = x$

A.  $-\infty$  and  $+\infty$

B.  $+0$  and  $-0$

C.  $-1$  and  $1$

D. A & B

E. A & C

# NaN (Not a Number)

- Invalid exception
  - Exact result is not a well-defined real number

$0/0$     $\sqrt{-1}$     $\infty - \infty$     $\text{NaN} - 10$     $\text{NaN} < 2?$

- We can have a quiet NaN or a signaling Nan
  - Quiet –does not raise an exception, but propagates a distinguished value
    - E.g. missing data:  $\max(3, \text{NaN}) = 3$
  - Signaling - generate an exception when accessed
    - Detect uninitialized data

What is the value of the expression NaN<2?

A. True

B. False

C. NaN

# Why are comparisons with NaN different from other numbers?

- Why does `NaN < 3` yield `false` ?
- Because `NaN` is not ordered with respect to any value
- Clause 5.11, paragraph 2 of the 754-2008 standard:  
4 mutually exclusive relations are possible:  
less than, equal, greater than, and unordered  
The last case arises when at least one operand is NaN.  
Every NaN shall compare unordered with everything,  
including itself
- See Stephen Canon's entry dated [2/15/09@17.00 on stackoverflow.com/questions/1565164/what-is-the-rationale-for-all-comparisons-returning-false-for-ieee754-nan-values](https://stackoverflow.com/questions/1565164/what-is-the-rationale-for-all-comparisons-returning-false-for-ieee754-nan-values)

# Working with NaNs

- A NaN is unusual in that you cannot compare it with anything, including itself!

```
#include<stdlib.h>
#include<iostream>
#include<cmath>
using namespace std;
int main() {
    float x = 0.0 / 0.0;
    cout << "0/0 = " << x << endl;
    if (std::isnan(x))
        cout << "IsNan\n";
    if (x != x)
        cout << "x != x is another way of saying isnan\n";
    return 0;
}
```

0/0 = nan

IsNan

x != x is another way of saying isnan

# Summary of representable numbers

- $\pm 0$

|       |       |         |
|-------|-------|---------|
| $\pm$ | 0...0 | 0.....0 |
|-------|-------|---------|

- $\pm \infty$

|       |        |         |
|-------|--------|---------|
| $\pm$ | 1....1 | 0.....0 |
|-------|--------|---------|

- Normalized nonzeros

|       |                    |          |
|-------|--------------------|----------|
| $\pm$ | Not 0 or<br>all 1s | anything |
|-------|--------------------|----------|

- Denormalized numbers

|       |       |         |
|-------|-------|---------|
| $\pm$ | 0...0 | nonzero |
|-------|-------|---------|

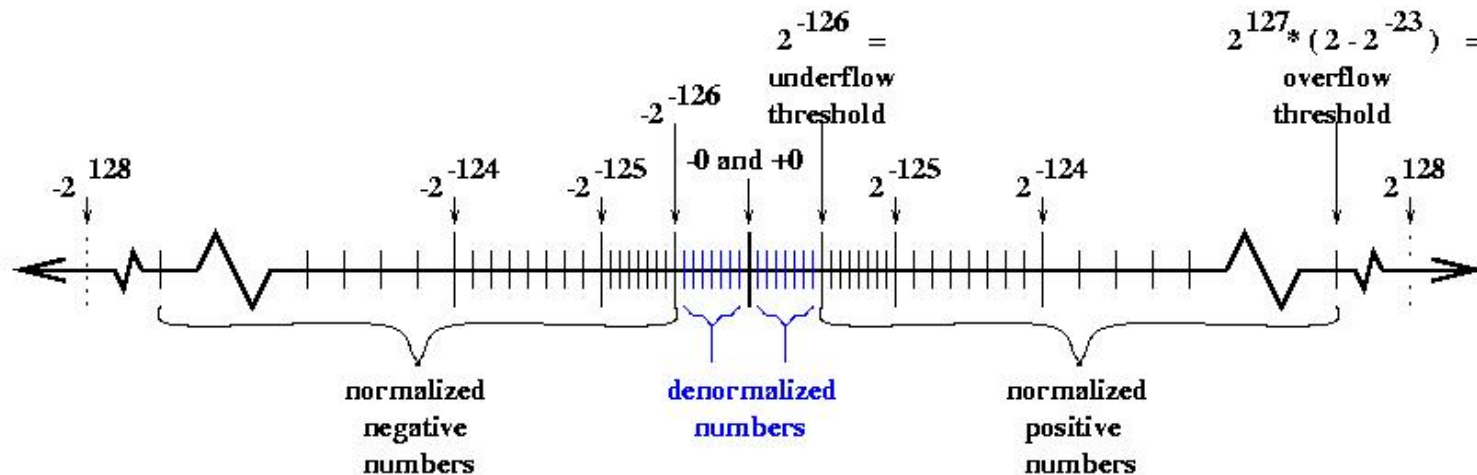
- NaNs

|       |       |         |
|-------|-------|---------|
| $\pm$ | 1...1 | nonzero |
|-------|-------|---------|

- Signaling and quiet  
(Signaling has most significant mantissa bit set to 0  
Quiet, the most significant bit set to 1)
  - Often supported as quiet NaNs only

# Denormalized numbers

- Let's compute: if  $(a \neq b)$  then  $x = a/(a-b)$
- We should never divide by 0, even if  $a-b$  is tiny
- *Underflow* exception occurs when exact result  $a-b < \text{underflow threshold UN}$  (too small to represent)
- We return a *denormalized number* for  $a-b$ 
  - Relax restriction that leading digit is 1:  $\pm 0.d\dots d \times 2^{\text{min\_exp}}$
  - Fills in the gap between 0 and UN uniform distribution of values
  - Ensures that we never divide by 0
  - Some loss of precision



Jim

## Extra precision

- The extended precision format provides 80 bits
- Enables us to reduce rounding errors
- Not obligatory, though many vendors support it
- We see extra precision in registers only
- There is a loss of precision when we store to memory (80  $\rightarrow$  64 bits)
- Not supported in SSE, scalar values only

| Format | # bits    | #significand bits | macheps                            | #exponent bits | exponent range                                  |
|--------|-----------|-------------------|------------------------------------|----------------|-------------------------------------------------|
| Single | 32        | 23+1              | $2^{-24}$ ( $\sim 10^{-7}$ )       | 8              | $2^{-126} - 2^{127}$ ( $\sim 10^{+-38}$ )       |
| Double | 64        | 52+1              | $2^{-53}$ ( $\sim 10^{-16}$ )      | 11             | $2^{-1022} - 2^{1023}$ ( $\sim 10^{+-308}$ )    |
| Double | $\geq 80$ | $\geq 64$         | $\leq 2^{-64}$ ( $\sim 10^{-19}$ ) | $\geq 15$      | $2^{-16382} - 2^{16383}$ ( $\sim 10^{+-4932}$ ) |

## When compiler optimizations alter precision

- If we support 80 bits extended format in registers..
- When we store values into memory, values will be truncated to the lower precision format, e.g. 64 bits
- Compilers can keep things in registers and we may lose *referential transparency*, depending on the optimization
- Example: `round(4.55,1)` should return 4.6, but returns 4.5 with `-O1` through `-O3`

```
double round(double v, double digit) {  
    double p = std::pow(10.0, digit);  
    double t = v * p;  
    double r = std::floor(t + 0.5);  
    return r / p; }
```

- With optimization turned on, `p` is computed to extra precision; it is not stored as a float (and rounded to 4.55), but lives in a register and is stored as 4.550000190734863
  - ▶ `t = v*p = 45.5000019073486`
  - ▶ `r = floor(t+45) = 46`
  - ▶ `r/p = 4.6`

# Exception handling - interface

- P754 standardizes how we handle exceptions
  - ▶ **Overflow**: - exact result  $> OV$ , too large to represent, returns  $\pm\infty$
  - ▶ **Underflow**: exact result nonzero and  $< UN$ , too small to represent
  - ▶ **Divide-by-zero**: nonzero/0, returns  $\pm\infty = \pm 0$  (Signed zeroes)
  - ▶ **Invalid**: 0/0,  $\sqrt{-1}$ ,  $\log(0)$ , etc.
  - ▶ **Inexact**: there was a rounding error (common)
- Each of the 5 exceptions manipulates 2 flags: should a trap occur?
  - ▶ By default we don't trap, but we continue computing  
**NaN**       $\infty$       **denorm**
  - ▶ If we do trap: enter a trap handler, we have access to arguments in operation that caused the exception
  - ▶ Requires precise interrupts, causes problems on a parallel computer, usually not implemented
- We can use exception handling to build faster algorithms
  - ▶ Try the faster but “riskier” algorithm (but denorm can be slow)
  - ▶ Rapidly test for accuracy (use exception handling)
  - ▶ Substitute slower more stable algorithm as needed
  - ▶ See Demmel&Li: [crd.lbl.gov/~xiaoye](http://crd.lbl.gov/~xiaoye)

Fin