CME213/ME339
Lecture 11

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Floating Point Mess

My results don't match!

- It is possible for CPU and GPU to both be "right" and still produce different results
- Often even parallel CPU code will produce different results than serial CPU code
- There exist many subtleties about floating point that can cause these numerical differences
- Knowing these differences exist does NOT give you freedom to declare that any difference is simply due to "floating point subtleties"
- You should be able to demonstrate where, why and how these numerical differences arise
- Often times this process will lead you to bugs

Points about Floating Point

Format

	Sign	Exponent	Mantissa
Single	1 bit	8 bits	23 bits
Double	1 bit	11 bits	52 bits

- $\pm Mantissa * 2^{exponent}$
- Sign bit is 0 for positive, 1 for negative
- The Mantissa is *always** in the range $[1, 2)$
- Make the leading 1 implicit, gaining us an extra bit
- Exponent is *biased* by 127 (1023 for double)
- -126 $=$ 1, 0 $=$ 127

Points about Floating Point

Examples

- To represent 192 is single precision floating point:
- It is positive - the sign bit is 0
- It is between 2^7 and 2^8 so the exponent is $7 + 127 = 134$
- $= 1.5 * 2^7$ so the Mantissa is 1.5 (remember the leading 1 is implicit in the binary format)

0	10000110	.10000000000000000000000

Points about Floating Point

Properties

- Is commutative $a + b = b + a$
- Is NOT associative $(a + b) + c \neq a + (b + c)$
- Is NOT distributive $a * (b + c) \neq a * b + a * c$
- $a - b = 0$ may not even imply that $a == b$
- $a = 1.01 * 2^{-126}$ and $b = 1 * 2^{-126}$, then $a - b = .01 * 2^{-126} = 1 * 2^{-128}$
- but -128 is not an allowable exponent and without denormal or subnormal numbers would be flushed to zero

Points about Floating Point

Subnormal Numbers

- Subnormal (sometimes known as denormal) numbers exist to ensure that $a - b = 0$ does imply that $a == b$
- Numbers with exponents < -126 instead set the exponent to all 0s
- Then the exponent is assumed to be -126 and the leading digit is now assumed to be 0
- So we could represent $.01 * 2^{-126} = 1 * 2^{-128}$ as follows

0	00000000	.01000000000000000000000

Can be controlled with the `nvcc` flag `-ftz=true|false`

Non-Associativity

Example - Binary Arithmetic

$$
\begin{aligned}
A &= 2^1 \times 1.00000000000000000000001 \\
B &= 2^0 \times 1.00000000000000000000001 \\
C &= 2^3 \times 1.00000000000000000000001
\end{aligned}
$$

Let's compute $(A + B) + C$ and $A + (B + C)$

$$
\begin{aligned}
A + B &= 2^1 \times 1.1000000000000000000000110000\ldots \\
rn(A + B) &= 2^1 \times 1.10000000000000000000010 \\
B + C &= 2^3 \times 1.0010000000000000000000100100\ldots \\
rn(B + C) &= 2^3 \times 1.00100000000000000000001 \\
A + B + C &= 2^3 \times 1.0110000000000000000000101100\ldots \\
rn(rn(A + B) + C) &= 2^3 \times 1.01100000000000000000010 \\
rn(A + rn(B + C)) &= 2^3 \times 1.01100000000000000000001
\end{aligned}
$$

Fused Multiply Add

FMA

- Allows for computing $A * B + C$ with only one round without rounding the result of $A * B$
- Generally leads to more accurate results
- An example with decimal arithmetic, 5 digits of precision:

$$\begin{aligned} x &= 1.0008 \\ x^2 &= 1.00160064 \\ x^2 - 1 &= 1.60064 \times 10^{-4} \\ rn(x^2 - 1) &= 1.6006 \times 10^{-4} \\ rn(x^2) &= 1.0016 \\ rn(rn(x^2) - 1) &= 1.6000 \times 10^{-4} \end{aligned}$$

Fused Multiply Add

Impact

- Usually FMAs are a significant source of the numerical differences between x86 and CUDA code
- Currently almost no x86 CPUs support this instruction which means that often the GPU results are *more* accurate
- You can force CUDA code to not combine multiplies and additions/subtractions into FMAs by explicitly specifying addition and multiplication
- $a + b$: `__fadd_rn(a, b)`
- $a * b$: `__fmul_rn(a, b)`

How to Measure Error?
ULP

- We use "Units in the last place" or ULPs to measure error
- The best possible error bound is .5 ULP
- 10000.5 $\rightarrow$ 10001
- All basic arithmetic operations are guaranteed to have this bound
- Compositions of these operations may result in significantly worse errors
- `__fmaf_rn(1.0008, 1.0008, -1)` has an error of .4 ULP
- `__fadd_rn(-1, __fmul_rn(1.0008, 1.0008))` has an error of 6.4 ULP!
- Appendix C of the NVIDIA Programming Guide has a list of ULP errors for all math library functions
  - Their convention is slightly different - they give ULP difference from the correctly rounded result not from the exact answer

ULP Errors

and the Table-Maker's Dilemma

Function	ULP Error
`sinf(x)`	2 (full range)
`powf(x, y)`	8 (full range)
`lgammaf(x)`	6 outside interval $[-10.001, -2.264]$ larger inside
`__sinf(x)`	for x in $[-\pi, \pi]$ maximum absolute error is $2^{-21.41}$ and larger otherwise

- "No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits. Even the fact (if true) that a finite number of extra digits will ultimately suffice may be a deep theorem." - William Kahan

Reductions and Floating Point

- Defined as $a_1 \oplus a_2 \oplus a_3 \oplus \dots$
- $\oplus$ can be any commutative and associative operation - $+, *, max, ...$
- But you just said floating point isn't associative!
  - True, but we do it anyway aware that changing the order of operations will change the result

Serial Implementation:

```
float sum = 0.f;
for (int i = 0; i < N; ++i)
    sum += vals[i];
```

The error associated with this summation grows is $O(\sqrt{N})$

A Better Way

- By changing the order of the summation, we can do a lot better without doing any more work
- Use a tree!
- Ex: $a_0 + a_1 + a_2 + a_3$

$$
\begin{aligned}
b_0 &= a_0 + a_1 \\
b_1 &= a_2 + a_3 \\
sum &= b_0 + b_1
\end{aligned}
$$

- Terms that are summed tend to be approximately equal in magnitude
- Leads to an error bound of $O(\sqrt{logN})$

Code for Parallel / Pairwise Summation

```
for (int i = 0; i < log2(N) + 1; ++i) {
    int offset = 1 << i;
    for (int j = 0; j < N; j += 2 * offset) {
        vals[j] += vals[j + offset];
    }
}
//sum is in vals[0]
```

0	1	2	3	4	5	6	7
1	1	5	3	9	5	13	7
6	1	5	3	22	5	13	7
28	1	5	3	22	5	13	7

Futher Resources

What Every Computer Scientist Should Know About Floating-Point Arithmetic `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`