Lecture 10

Performance programming for stencil methods
Floating point arithmetic
Announcements

• Lincoln
  ♦ Each mush have an individual account

• Calendar
  ♦ Triton/SDSC workshop and visit (Lecture 13)
    • Weds Nov 3rd: 3:30 to 5:00
  ♦ Midterm: Tuesday November 9th
  ♦ Midterm return and lecture 17 on Friday Nov 12 (4p to 5:20p)
  ♦ No class on Tuesday/Thursday Nov 16th/18th
Today’s lecture

- More Performance Programming
  - Stencil Methods
  - Matrix multiplication
- Floating point arithmetic
Stencil Methods
Stencil Methods – using ghost cells

Rivera and Tseng
Floating Point Arithmetic
What is floating point?

• A representation
  ⊲ ±2.5732… × 10^{22}
  ⊲ NaN ∞
  ⊲ Single, double, extended precision

• A set of operations
  ⊲ + = * / √ rem
  ⊲ Comparison < ≤ = ≠ ≥>
  ⊲ Conversions between different formats, binary to decimal
  ⊲ Exception handling

• IEEE Floating point standard P754
  ⊲ Universally accepted
  ⊲ W. Kahan received the Turing Award in 1989 for design of IEEE Floating Point Standard
  ⊲ Revision in 2008
IEEE Floating point standard P754

- Normalized representation  \( \pm 1.d\ldots d \times 2^{\text{esp}} \)
  - Macheps = Machine epsilon = \( \varepsilon = 2^{-\#\text{significand bits}} \)
  - OV = overflow threshold = largest number
  - UN = underflow threshold = smallest number
- \( \pm \text{Zero} \): \( \pm \text{significand and exponent } = 0 \)

<table>
<thead>
<tr>
<th>Format</th>
<th># bits</th>
<th>#significand bits</th>
<th>macheps</th>
<th>#exponent bits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>23+1</td>
<td>( 2^{-24} (\sim 10^{-7}) )</td>
<td>8</td>
<td>( 2^{-126} - 2^{127} (\sim 10^{+38}) )</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>52+1</td>
<td>( 2^{-53} (\sim 10^{-16}) )</td>
<td>11</td>
<td>( 2^{-1022} - 2^{1023} (\sim 10^{+308}) )</td>
</tr>
<tr>
<td>Double</td>
<td>( \geq 80 )</td>
<td>( \geq 64 )</td>
<td>( \leq 2^{-64} (\sim 10^{-19}) )</td>
<td>( \geq 15 )</td>
<td>( 2^{-16382} - 2^{16383} (\sim 10^{+4932}) )</td>
</tr>
</tbody>
</table>
What happens in a floating point operation?

- Round to the nearest representable floating point number that corresponds to the exact value (correct rounding).
- Round to nearest value with the lowest order bit = 0 (rounding toward nearest even).
- Others are possible.
- We don’t need the exact value to work this out!
- Applies to + = * / √ rem.

Error formula: \( \text{fl}(a \ op \ b) = (a \ op \ b)*(1 + \delta) \) where
  - \( \delta \) one of + , - , * , /
  - \(| \delta | \leq \varepsilon\)
    - assuming no overflow, underflow, or divide by zero.

Addition example:
  - \( \text{fl}(\sum x_i) = \sum_{i=1:n} x_i *(1+e_i) \)
  - \(|e_i| \sim< (n-1)\varepsilon\)
Exception Handling

• An exception occurs when the result of a floating point operation is not representable as a normalized floating point number
  - 1/0, √-1

• P754 standardizes how we handle exceptions
  - **Overflow**: exact result > OV, too large to represent
  - **Underflow**: exact result nonzero and < UN, too small to represent
  - **Divide-by-zero**: nonzero/0
  - **Invalid**: 0/0, √-1, log(0), etc.
  - **Inexact**: there was a rounding error (common)

• Two possible responses
  - Stop the program, given an error message
  - Tolerate the exception
An example

• Graph the function

\[ f(x) = \frac{\sin(x)}{x} \]

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

- We compute \(\text{if } (a \neq b) \text{ then } x = a/(a-b)\)
- We should never divide by 0, even if \(a-b\) is tiny
- Underflow exception occurs when exact result \(a-b < \text{underflow threshold \text{ UN}}\)
- We return a denormalized number for \(a-b\)
  - \(\pm0.d\ldots d \times 2^{\text{min}\_\text{exp}}\)
  - sign bit, nonzero significand, minimum exponent value
  - Fill in the gap between 0 and \text{UN}
NaN (Not a Number)

- **Invalid exception**
  - Exact result is not a well-defined real number
- **We can have a quiet NaN or an sNan**
  - Quiet – does not raise an exception, but propagates a distinguished value
    - E.g. missing data: max(3, NAN) = 3
  - Signaling - generate an exception when accessed
    - Detect uninitialized data
Exception handling

• Each of the 5 exceptions manipulates 2 flags
• Sticky flag set by an exception, can be read and cleared by the user
• Exception flag: should a trap occur?
  ♦ If so, we can enter a trap handler
  ♦ But requires precise interrupts, causes problems on a parallel computer
• We can use exception handling to build faster algorithms
  ♦ Try the faster but “riskier” algorithm
  ♦ Rapidly test for accuracy (possibly with the aid of exception handling)
  ♦ Substitute slower more stable algorithm as needed
When compiler optimizations alter precision

- Let’s say we support 79+ bit extended format in registers
- When we store values into memory, values are converted to the lower precision format
- Compilers can keep things in registers and we may lose referential transparency
- An example
  ```
  float x, y, z;
  int j;
  ....
  x = y + z;
  if (x >= j) replace x by something smaller than j
  y=x;
  ```
- With optimization turned on, x is computed to extra precision; it is not a float
- If x < j in a register, there is no guarantee the condition will be preserved when x is stored in y, i.e. y >= j
P754 on the GPU

- Cuda Programming Guide (3.2)
  “All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations”
  - There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the exceptions are always masked… SNaN … are handled as quiet
- For single-precision floating-point numbers on devices of compute capability 1.x:
  - Denormalized numbers are not supported; floating-point arithmetic and comparison instructions convert denormalized operands to zero prior to the operation
  - Results that underflow are flushed to zero
- Some instructions are not IEEE-compliant:
  - + and * are combined into a single multiply-add instruction (FMAD), which truncates (i.e. without rounding) the intermediate mantissa of the multiplication
  - Division is implemented via the reciprocal in a non-standard-compliant way
- Things change for device capability 2.x

©2010 Scott B. Baden /CSE 260/ Fall '10