Floating Point Issues

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

Outline

- Floating Point Basics
- Floating-Point Computing in CUDA
Floating Point Basics

Integer Binary Representation

- Decimal
  2761050404

- Binary:
  1010 0100 1001 0010 0100 1001 0010 0100

- Hexadecimal
  A4924924

- Converting Binary to Decimal:
  \[ 1 \times 2^{31} + 1 \times 2^{29} + \ldots + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^2 \]
Signed Integer Representation

- Two’s Complement
  \[1010 \ 0100 \ 1001 \ 0010 \ 0100 \ 1001 \ 0010 \ 0100_2\]
  \[-1 \times 2^{31} + 1 \times 2^{29} + \ldots + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^2\]

- What’s the binary representation of -1?
  \[1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_2\]

Floating Point Numbers

- Real numbers
  \[3.141519 \quad 6.02 \times 10^{25}\]

- Floating Point binary representation:
  \[101.01011\]
  \[1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}\]

- Binary representation is finite, real numbers are infinite
  - A subset of the numbers can be represented
    - Limited range
    - Limited precision
Normalization

- Also known as scientific representation

\[ 6.02 \times 10^{25} \]

- Exactly one digit to the left of the decimal point

- Normalized binary representation

\[ 1.0011001001 \times 2^{23} \]

No zeros to the left of the decimal point, exactly one 1

---

IEEE 754 Standard

- IEEE Standard 754 for Binary Floating-Point Arithmetic
  - First standardized in 1985
  - Current version IEEE754-2008

- Architect: William Kahan
  - 1989 ACM Turing Award Winner!

- Read the story in this interview:
IEEE 754 Standard

- Arithmetic Formats:
  - Sets of binary and decimal floating-point data, which consist of finite numbers, infinities, and special 'not a number' values (NaNs)

- Operations:
  - Arithmetic and other operations on arithmetic formats

- Rounding Algorithms:
  - Methods to be used for rounding numbers during arithmetic and conversions

- Exception Handling:
  - Indications of exceptional conditions (such as division by zero, overflow, etc.)

---

Single-Precision FP

The single-precision floating-point format consists of:

- 1 sign bit (S)
- 8 bits exponent (E)
- 23 bits fraction (F)

The value is calculated as:

\[ \text{value} = (-1)^S \times (1.F) \times 2^{(E-127)} \]

**Example:**

- sign = 1
- exponent = 73
- fraction = \(1 \times 2^3 + 1 \times 2^6 + \ldots + 1 \times 2^{-15} + 1 \times 2^{-16} + 1 \times 2^{-21} = 0.142857074\)
- value = \(-1.142857074 \times 2^{-54}\)

*float type in C*
Single-Precision Floating Point

What’s the representation of $0.2_{10}$ in the IEEE-754 SP Floating Point?

- **Step 1:** convert 0.2 fraction to binary
  
  $0.2 \times 2 = 0.4$ \hspace{2em} $0.4 \times 2 = 0.8$ \hspace{2em} $0.8 \times 2 = 1.6$ \hspace{2em} $0.6 \times 2 = 1.2$
  
  $0.2 \times 2 = 0.4$ \hspace{2em} $0.4 \times 2 = 0.8$ \hspace{2em} $0.8 \times 2 = 1.6$ \hspace{2em} $0.6 \times 2 = 1.2$ ...

  \[\text{fraction} = 0011001100110011001\ldots\]
  
  normalized $= 1.100110011001100110011001100 \times 2^{-3}$

- **Step 2:** calculate the exponent
  
  $\text{exponent} = -3 + 127 = 124 = 01111100$

- **Step 3:** Single-Precision Format:
  
  \[
  \begin{array}{cccccccc}
  0 & 011 & 1110 & 0 & 100 & 1100 & 1100 & 1100 & 1100 \\
  \end{array}
  \]

Overflow, Underflow, and Precision

- **Overflow**
  
  - Largest SP FP: $1 \times 2^{128} = 2.0 \times 10^{38}$
  - What if $(a \times b) > 2.0 \times 10^{38}$
  - Exponent larger than 128 (max represented in 8-bit exponent field)

- **Underflow**
  
  - Smallest number (absolute value) in SP: $1 \times 2^{-127} = 2.0 \times 10^{-38}$
  - What if $0 < (a \times b) < 2.0 \times 10^{-38}$
  - Exponents smaller than -127

- **Precision**
  
  - 23 bits for fraction (around 7 decimal digits)
  - There may be loss of precisions in FP arithmetic if the intermediate results are less than $2^{-23}$
Double-Precision FP

![Double-Precision FP Diagram]

\[
\text{value} = (-1)^S \times (1.\text{fraction}) \times 2^{(\text{exponent} - 1023)}
\]

**Double Precision Range:**

\[
2.22507 \times 10^{-308} \ldots 1.76769 \times 10^{308}
\]

**Precision:** \(2^{51}\) which is equal to about 16 decimal digit

---

Special Bit Sequences: Zero

**Single Precision**

- \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ = +0\) (or 0)
- \(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ = -0\)

**Double Precision**

- \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ = +0\)
- \(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ = -0\)

- Zero is signed
- No 1.xxx… mantissa is implied here.
Special Bit Sequences: Infinity

Single Precision

| 0111 1111 1000 0000 0000 0000 0000 0000 | = +∞ 
| 1111 1111 1000 0000 0000 0000 0000 0000 | = -∞

Double Precision

| 0111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 | = +∞ 
| 1111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 | = -∞

- ∞ = 1 / +0
- ∞ = 1 / -0

- Both operations are accompanied by division by zero exception
- Some operations on infinity:
  - +∞ + 7 = +∞
  - +∞ * -2 = -∞
  - +∞ * 0 = NaN

Special Bit Sequences: NaN

- Not a Number

Single Precision

| 0111 1111 11xx xxxx xxxx xxxx xxxx | Signaling NaN
| 1111 1111 10xx xxxx xxxx xxxx xxxx | Quite NaN

- How to get a NaN
  - +∞ * 0 = NaN
  - sqrt(-1) = NaN
  - 0 / 0 = NaN

- Signaling NaN
  - Any operation on it results in an “invalid” exception
- Quite NaN
  - Any operation on it results in NaN
Subnormal (Denormalized) Numbers

- **Motivation**
  - The smallest number we can show is: 
    \[ a = 1.0000000000\ldots \times 2^{-126} \]
  - The second smallest number is: 
    \[ b = 1.000\ldots1 \times 2^{-126} = a + 2^{-149} \]
  - \( b - a = 2^{-149} \) but \( a - 0 = 2^{-126} \)
  - The problem: normalization requirement

- Normalization requirement is relaxed for small numbers
  - Whenever the **exponent = 0**, the mantissa is no longer assumed to be of the form 1.xxx..., rather, it is assumed to be of the form 0.xxx.

  - So, if the n-bit exponent is 0, the value is
    \[ \text{value} = (-1)^S \times (0.\text{fraction}) \times 2^{(\text{exponent} - 127)} \]

Summary of the FP Encodings

- Infinity: \(-\infty\) and \(+\infty\)
- Normalized
- Denorm
- +Denorm
- +Normalized

\[ \text{slide from course 15-213 at CMU} \]
Floating Point Addition Example

- For simplicity
  - base=10
  - fraction is 3 digits
  - exponent is 1 digit

- Let's calculate this expression:
  - 99.98 + 0.226
  - we expect the result to be: 100.206

- Need to represent the numbers in normalized form:
  - $9.998 \times 10^1 + 2.26 \times 10^{-1}$

---

Floating Point Addition Example

- Step 1: equalize the exponents
  - $9.998 \times 10^1 + 0.022 \times 10^1$

- Step 2: add the mantissa
  - $9.998 \times 10^1 + 0.022 \times 10^1 = 10.020 \times 10^1$

- Step 3: Normalized the result
  - $1.002 \times 10^2$

- The error: $0.006$
  - Due to limited precision
Rounding Modes

- **Round towards** + $\infty$
  - ALWAYS round “up”: $2.1 \Rightarrow 3, -2.1 \Rightarrow -2$

- **Round towards** - $\infty$
  - ALWAYS round “down”: $1.9 \Rightarrow 1, -1.9 \Rightarrow -2$

- **Round towards 0** (i.e., truncate)
  - Just drop the last bits

- **Round half to the nearest integer** (default)
  - Rounding half to the nearest even: $2.5 \Rightarrow 2, 3.5 \Rightarrow 4$
  - Insures fairness on calculation
  - Half the time we round up, other half down

FP Multiplication

\[
\begin{align*}
A &= (-1)^{s_1} \times M_1 \times 2^{E_1} \\
B &= (-1)^{s_2} \times M_2 \times 2^{E_2} \\
C &= A \times B \\
C &= (-1)^s \times M \times 2^E
\end{align*}
\]

- **Sign** $s = s_1 \lor s_2$ (exclusive or)
- **Mantissa** $M = M_1 \times M_2$
- **Exponent** $E = E_1 + E_2$

- Normalizing
  - If $M \geq 2$, shift $M$ right, increment $E$
  - If $E$ out of range, overflow
  - Round $M$ to fit fraction precision
FP Addition

\[
A = (-1)^{s_1} \times M_1 \times 2^{E_1}
\]
\[
B = (-1)^{s_2} \times M_2 \times 2^{E_2}
\]
\[
C = A + B = (-1)^{s} \times M \times 2^{E}
\]

- **s = s_1**
- **E = E_1**

- **Normalizing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(M < 1\), shift \(M\) left \(k\) positions, decrement \(E\) by \(k\)
  - Overflow if \(E\) out of range
  - Round \(M\) to fit \(\text{frac}\) precision

---

Mathematical Properties of FP Addition

- **Closed under addition? YES**
  - But may generate infinity or NaN

- **Commutative? YES**

- **Associative \(a+(b+c) = (a+b)+c\)? NO**
  - Overflow and inexactness of rounding

- **0 is additive identity \(a+0=a\)? YES**

- **Every element has additive inverse? YES**
  - Except for infinities & NaNs

- **Montonicity \(a \geq b \Rightarrow a+c \geq b+c\)? YES**
  - Except for infinities & NaNs
Algebraic Properties of FP Multiplication

- Closed under multiplication? YES
  - But may generate infinity or NaN
- Commutative? YES
- Associative \( a^* (b^* c) = (a^* b)^* c \)? NO
  - Possibility of overflow, inexactness of rounding
- 1 is multiplicative identity \( a^* 1 = a \)? YES
- Multiplication distributes over addition \( a^* (b + c) = a^* b + a^* c \)? NO
  - Possibility of overflow, inexactness of rounding
- Montonicity \( a \geq b \text{ & } c \geq 0 \Rightarrow a^* c \geq b^* c \)? YES
  - Except for infinities & NaNs

Floating Point Computation in CUDA
Parallel vs. Sequential Code

- Addition and Multiplication are not associative.

Example:
- Let’s add the following numbers $1, 1, \frac{1}{4}, \frac{1}{4}$
- Let’s assume a 5-bit floating point: 1-bit sign, 2-bit fraction, 2-bit exponent
- Floating Point representation: $1.00 \times 2^0, 1.00 \times 2^0, 1.00 \times 2^{-2}, 1.00 \times 2^{-2}$

Sequential Execution

[Diagram of sequential execution of numbers]
Parallel Execution

\[
\begin{align*}
1.00 \times 2^0 & \quad 1.00 \times 2^0 \\
1.00 \times 2^1 & \quad 1.00 \times 2^1 \\
1.00 \times 2^{-1} & \quad 1.00 \times 2^{-1} \\
1.01 \times 2^1 & 
\end{align*}
\]

Sequential: 1.00 \times 2^1 \text{ Error=0.01}

Deviations from IEEE-754

- Subnormal (Denormalized) numbers are not supported
- Addition and Multiplication are IEEE 754 compliant
  - Maximum 0.5 ULP (Units in the Least Place) error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated
- Division is non-compliant (2 ULP)
  - Implemented using iterative approximation
  - Smaller number of iterations than CPU
- Not all rounding modes are supported
- No mechanism to detect floating-point exceptions
Units in the Least (Last) Place (ULP)

- The absolute value of the difference between the two numbers in a given finite numerical representation which are closest to any given number.

- Example:
  - Let's assume a 5-bit FP representation
  - Let's assume the result is 0.03123
  - ULP for this number is:
    - $3.13 \times 10^{-2} - 3.12 \times 10^{-2} = 0.0001$
  - Error for rounding 0.03123 toward infinity = 0.7 ULP
  - Error for rounding 0.03123 toward 0 = 0.3 ULP
  - Max error for rounding to the nearest number = 0.5 ULP

---

Double-Precision on GPU?

- GT200 has double precision support
  - G80 is single-precision only
  - Double precision has additional performance cost
    - Only one unit per multiprocessor
  - Careless use of double or undeclared types may run more slowly on G80+

<table>
<thead>
<tr>
<th>Architecture</th>
<th>FP Support</th>
<th>Single-Precision Speed (Max)</th>
<th>Double-Precision Speed (Max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G80</td>
<td>Single</td>
<td>400 GFLOPS</td>
<td>30 GFLOPS</td>
</tr>
<tr>
<td>GT200</td>
<td>Double</td>
<td>933 GFLOPS</td>
<td>78 GFLOPS</td>
</tr>
<tr>
<td>Fermi (GTX 400)</td>
<td>Double</td>
<td>1600 GFLOPS</td>
<td>630 GFLOPS</td>
</tr>
</tbody>
</table>
Make Your Program Float Safe

- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed

- Add ‘f’ specifier on float literals:
  - `foo = bar * 0.123;` // double assumed
  - `foo = bar * 0.123f;` // float explicit

- Use float version of standard library functions
  - `foo = sin(bar);` // double assumed
  - `foo = sinf(bar);` // single precision explicit

---

Arithmetic Instruction Throughput

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Operation</th>
<th>Latency Per Warp (cycles)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>add, shift, min, max</td>
<td>4</td>
<td>hardware support</td>
</tr>
<tr>
<td>int</td>
<td>multiply</td>
<td>many cycles</td>
<td>32-bit by default</td>
</tr>
<tr>
<td>int</td>
<td>__mul24(), __umul24()</td>
<td>4</td>
<td>24-bit multiplication intrinsics provided by CUDA</td>
</tr>
<tr>
<td>float</td>
<td>add, mul, mad</td>
<td>4</td>
<td>single-precision (double precision takes many more cycles)</td>
</tr>
<tr>
<td>int</td>
<td>divide, modulo</td>
<td>many cycles</td>
<td>use shift and and for power-of-2 divide and module instead: a/2 == a&gt;&gt;1, a%n == a&amp;(n-1) compiler automatically converts a/2 to a&gt;&gt;1</td>
</tr>
</tbody>
</table>

---

Published by GPU Programming Shiraz University, Winter 88/Spring 89, Reza Azimi
### Arithmetic Instruction Throughput

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Operation</th>
<th>Latency Per Warp (cycles)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>reciprocal</td>
<td>16</td>
<td>syntax: __rcp()</td>
</tr>
<tr>
<td>float</td>
<td>reciprocal, square root</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>sin, cos</td>
<td>16</td>
<td>syntax: __sin(), __cos()</td>
</tr>
<tr>
<td>float</td>
<td>exp</td>
<td>16</td>
<td>syntax: __exp</td>
</tr>
</tbody>
</table>

Other functions are combinations of the above:
- \( y / x = \text{rcp}(x) \times y \) : 20 cycles per warp
- \( \sqrt{x} = \text{rcp}(\text{rsqrt}(x)) \) : 32 cycles per warp

---

### Runtime Math Library

- There are two types of runtime math operations
  - __func() : direct mapping to hardware ISA
    - Fast but low accuracy (see the NVIDIA CUDA Programmer’s Guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ULP)
    - Examples: sin(x), exp(x), pow(x,y)

- The -use_fast_math compiler option forces every func() to compile to __func()
### Summary of GPU Floating Point Features

<table>
<thead>
<tr>
<th></th>
<th>G80</th>
<th>SSE</th>
<th>IBM Altivec</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
<tr>
<td><strong>Rounding modes</strong>&lt;br&gt;for FADD and FMUL</td>
<td>Round to nearest and round to zero</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>Round to nearest only</td>
<td>Round to zero/truncate only</td>
</tr>
<tr>
<td><strong>Denormal handling</strong></td>
<td>Flush to zero</td>
<td>Supported, 1000’s of cycles</td>
<td>Supported, 1000’s of cycles</td>
<td>Flush to zero</td>
</tr>
<tr>
<td><strong>NaN support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Overflow and Infinity support</strong></td>
<td>Yes, only clamps to max norm</td>
<td>Yes</td>
<td>Yes</td>
<td>No, infinity</td>
</tr>
<tr>
<td><strong>Square root</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Reciprocal estimate accuracy</strong></td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>Reciprocal sqrt estimate accuracy</strong></td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>log2(x) and 2^x estimates accuracy</strong></td>
<td>23 bit</td>
<td>No</td>
<td>12 bit</td>
<td>No</td>
</tr>
</tbody>
</table>