VSCSE summer school - short course

Introduction to CUDA

# Lecture 7
# Floating point precision

Joshua A. Anderson

# Objective

- To understand the fundamentals of floating-point representation
- To know the IEEE-754 Floating Point Standard
- CUDA Floating-point speed, accuracy and precision
  - Deviations from IEEE-754
  - Accuracy of device runtime functions
  - -fastmath compiler option
  - Future performance considerations

# GPU Floating Point Features

| | CUDA | SSE | IBM Altivec | Cell SPE |
|---|---|---|---|---|
| Precision | IEEE 754 | IEEE 754 | IEEE 754 | IEEE 754 |
| Rounding modes for FADD and FMUL | All 4 IEEE, round to nearest, zero, inf, -inf | All 4 IEEE, round to nearest, zero, inf, -inf | Round to nearest only | Round to zero/ truncate only |
| Denormal handling | 1.x - Flush to zero<br>2.0 - Supported | Supported, 1000's of cycles | Supported, 1000's of cycles | Flush to zero |
| NaN support | Yes | Yes | Yes | No |
| Overflow and Infinity support | Yes, only clamps to max norm | Yes | Yes | No, infinity |
| Flags | No | Yes | Yes | Some |
| Square root | Software | Hardware | Software only | Software only |
| Division | Software | Hardware | Software only | Software only |
| Reciprocal estimate accuracy | 24 bit | 12 bit | 12 bit | 12 bit |
| Reciprocal sqrt estimate accuracy | 23 bit | 12 bit | 12 bit | 12 bit |
| log2(x) and 2^x estimates accuracy | 23 bit | No | 12 bit | No |

# What is IEEE floating-point format?

- A floating point binary number consists of three parts:
  - sign (S), exponent (E), and mantissa (M).
  - Each (S, E, M) pattern uniquely identifies a floating point number.

- For each bit pattern, its IEEE floating-point value is derived as:

  - value = $(-1)^S * M * \{2^E\}$, where $1.0 \leq M < 10.0_{\mathbf{B}}$

- The interpretation of S is simple: S=0 results in a positive number and S=1 a negative number.

# Normalized Representation

- Specifying that $1.0_B \le M < 10.0_B$ makes the mantissa value for each floating point number unique.
  - For example, the only one mantissa value allowed for $0.5_D$ is M =1.0
    - $0.5_D = 1.0_B * 2^{-1}$
  - Neither $10.0_B * 2^{-2}$ nor $0.1_B * 2^{0}$ qualifies

- Because all mantissa values are of the form 1.XX…, one can omit the "1." part in the representation.
  - The mantissa value of $0.5_D$ in a 2-bit mantissa is 00, which is derived by omitting "1." from 1.00.

# Exponent Representation

- In an n-bits exponent representation, $2^{n-1}-1$ is added to its 2's complement representation to form its excess representation.
  - See Table for a 3-bit exponent representation
- A simple unsigned integer comparator can be used to compare the magnitude of two FP numbers
- Symmetric range for +/- exponents (111 reserved)

| 2's complement | Actual decimal | Excess-3 |
|---|---|---|
| 000 | 0 | 011 |
| 001 | 1 | 100 |
| 010 | 2 | 101 |
| 011 | 3 | 110 |
| **100** | **(reserved pattern)** | **111** |
| 101 | -3 | 000 |
| 110 | -2 | 001 |
| 111 | -1 | 010 |

# A simple, hypothetical 5-bit FP format

- Assume 1-bit S, 2-bit E, and 2-bit M
  - $0.5_D = 1.00_B * 2^{-1}$
  - $0.5_D$ = **0 00 00**, where S = 0, E = 00, and M = (1.)00

| 2's complement | Actual decimal | Excess-1 |
|---|---|---|
| 00 | 0 | 01 |
| 01 | 1 | 10 |
| 10 | (reserved pattern) | 11 |
| 11 | -1 | **00** |

# Representable Numbers

- The representable numbers of a given format is the set of all numbers that can be exactly represented in the format.
- See Table for representable numbers of an unsigned 3-bit integer format


| | |
|---|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

# Represent … f a 5-bit Hypot … rmat

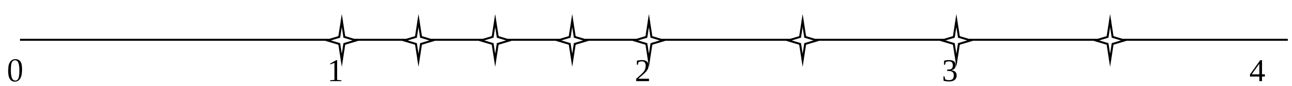
Cannot represent Zero!

0

| | | No-zero | | Abrupt underflow | | Gradual underflow | |
|---|---|---|---|---|---|---|---|
| E | M | **S=0** | **S=1** | S=0 | S=1 | S=0 | S=1 |
| 00 | 00 | **$2^{-1}$** | **$-(2^{-1})$** | 0 | 0 | 0 | 0 |
| | 01 | **$2^{-1}+1*2^{-3}$** | **$-(2^{-1}+1*2^{-3})$** | 0 | 0 | $1*2^{-2}$ | $-1*2^{-2}$ |
| | 10 | **$2^{-1}+2*2^{-3}$** | **$-(2^{-1}+2*2^{-3})$** | 0 | 0 | $2*2^{-2}$ | $-2*2^{-2}$ |
| | 11 | **$2^{-1}+3*2^{-3}$** | **$-(2^{-1}+3*2^{-3})$** | 0 | 0 | $3*2^{-2}$ | $-3*2^{-2}$ |
| 01 | 00 | **$2^{0}$** | **$-(2^{0})$** | $2^{0}$ | $-(2^{0})$ | $2^{0}$ | $-(2^{0})$ |
| | 01 | **$2^{0}+1*2^{-2}$** | **$-(2^{0}+1*2^{-2})$** | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ |
| | 10 | **$2^{0}+2*2^{-2}$** | **$-(2^{0}+2*2^{-2})$** | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ |
| | 11 | **$2^{0}+3*2^{-2}$** | **$-(2^{0}+3*2^{-2})$** | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ |
| 10 | 00 | **$2^{1}$** | **$-(2^{1})$** | $2^{1}$ | $-(2^{1})$ | $2^{1}$ | $-(2^{1})$ |
| | 01 | **$2^{1}+1*2^{-1}$** | **$-(2^{1}+1*2^{-1})$** | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ |
| | 10 | **$2^{1}+2*2^{-1}$** | **$-(2^{1}+2*2^{-1})$** | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ |
| | 11 | **$2^{1}+3*2^{-1}$** | **$-(2^{1}+3*2^{-1})$** | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ |
| 11 | Reserved pattern | | | | | | |

# Flush to Zero

- Treat all bit patterns with E=0 as 0.0
  - This takes away several representable numbers near zero and lump them all into 0.0
  - For a representation with large M, a large number of
0 representable numbers numbers will be removed.



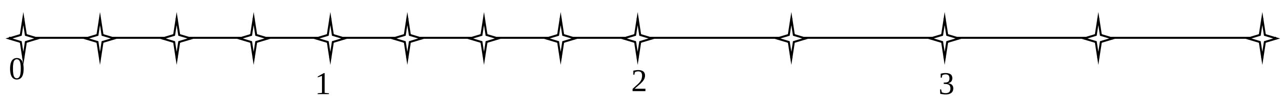
# Flush to Zero

0

| | | No-zero | | **Flush to Zero** | | Denormalized | |
|---|---|---|---|---|---|---|---|
| E | M | S=0 | S=1 | **S=0** | **S=1** | S=0 | S=1 |
| 00 | 00 | $2^{-1}$ | $-(2^{-1})$ | **0** | **0** | 0 | 0 |
| | 01 | $2^{-1}+1*2^{-3}$ | $-(2^{-1}+1*2^{-3})$ | **0** | **0** | $1*2^{-2}$ | $-1*2^{-2}$ |
| | 10 | $2^{-1}+2*2^{-3}$ | $-(2^{-1}+2*2^{-3})$ | **0** | **0** | $2*2^{-2}$ | $-2*2^{-2}$ |
| | 11 | $2^{-1}+3*2^{-3}$ | $-(2^{-1}+3*2^{-3})$ | **0** | **0** | $3*2^{-2}$ | $-3*2^{-2}$ |
| 01 | 00 | $2^{0}$ | $-(2^{0})$ | $\mathbf{2^{0}}$ | $\mathbf{-(2^{0})}$ | $2^{0}$ | $-(2^{0})$ |
| | 01 | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ | $\mathbf{2^{0}+1*2^{-2}}$ | $\mathbf{-(2^{0}+1*2^{-2})}$ | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ |
| | 10 | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ | $\mathbf{2^{0}+2*2^{-2}}$ | $\mathbf{-(2^{0}+2*2^{-2})}$ | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ |
| | 11 | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ | $\mathbf{2^{0}+3*2^{-2}}$ | $\mathbf{-(2^{0}+3*2^{-2})}$ | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ |
| 10 | 00 | $2^{1}$ | $-(2^{1})$ | $\mathbf{2^{1}}$ | $\mathbf{-(2^{1})}$ | $2^{1}$ | $-(2^{1})$ |
| | 01 | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ | $\mathbf{2^{1}+1*2^{-1}}$ | $\mathbf{-(2^{1}+1*2^{-1})}$ | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ |
| | 10 | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ | $\mathbf{2^{1}+2*2^{-1}}$ | $\mathbf{-(2^{1}+2*2^{-1})}$ | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ |
| | 11 | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ | $\mathbf{2^{1}+3*2^{-1}}$ | $\mathbf{-(2^{1}+3*2^{-1})}$ | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ |
| 11 | Reserved pattern | | | | | | |

# Denormalized Numbers

- The actual method adopted by the IEEE standard is called denromalized numbers or gradual underflow.
  - The method relaxes the normalization requirement for numbers very close to 0.
  - whenever E=0, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. In general, if the n-bit exponent is 0, the value is
    - $0.M * 2^{-2^{(n-1)}} + 2$



# Denormalization

0

| | | No-zero | | Flush to Zero | | **Denormalized** | |
|---|---|---|---|---|---|---|---|
| E | M | S=0 | S=1 | S=0 | S=1 | **S=0** | **S=1** |
| 00 | 00 | $2^{-1}$ | $-(2^{-1})$ | 0 | 0 | **0** | **0** |
| | 01 | $2^{-1}+1*2^{-3}$ | $-(2^{-1}+1*2^{-3})$ | 0 | 0 | **$1*2^{-2}$** | **$-1*2^{-2}$** |
| | 10 | $2^{-1}+2*2^{-3}$ | $-(2^{-1}+2*2^{-3})$ | 0 | 0 | **$2*2^{-2}$** | **$-2*2^{-2}$** |
| | 11 | $2^{-1}+3*2^{-3}$ | $-(2^{-1}+3*2^{-3})$ | 0 | 0 | **$3*2^{-2}$** | **$-3*2^{-2}$** |
| 01 | 00 | $2^{0}$ | $-(2^{0})$ | $2^{0}$ | $-(2^{0})$ | $2^{0}$ | $-(2^{0})$ |
| | 01 | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ | $2^{0}+1*2^{-2}$ | $-(2^{0}+1*2^{-2})$ |
| | 10 | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ | $2^{0}+2*2^{-2}$ | $-(2^{0}+2*2^{-2})$ |
| | 11 | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ | $2^{0}+3*2^{-2}$ | $-(2^{0}+3*2^{-2})$ |
| 10 | 00 | $2^{1}$ | $-(2^{1})$ | $2^{1}$ | $-(2^{1})$ | $2^{1}$ | $-(2^{1})$ |
| | 01 | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ | $2^{1}+1*2^{-1}$ | $-(2^{1}+1*2^{-1})$ |
| | 10 | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ | $2^{1}+2*2^{-1}$ | $-(2^{1}+2*2^{-1})$ |
| | 11 | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ | $2^{1}+3*2^{-1}$ | $-(2^{1}+3*2^{-1})$ |
| 11 | | Reserved pattern | | | | | |

# Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad (fma on Fermi)
  - Compute 1.x - 4 cycles per warp
  - Compute 2.0 – 1 cycle per warp
  - int multiply (*) is by default 32-bit
    - requires multiple instructions  on compute 1.x
    - Only 1 cycle on compute 2.x
  - Can Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply
    - But don't on compute 2.x – maps to multiple instructions
- Integer divide and modulo are expensive
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

# Arithmetic Instruction Throughput

- Reciprocal, reciprocal square root, sin/cos, log, exp
  - Compute 1.x - 16 cycles per warp
  - Compute 2.x – 8 cycles per warp
  - These are the versions prefixed with "__"
  - Examples:__rcp(), __sin(), __exp()

- Other functions are combinations of the above
  - y / x == rcp(x) * y == 20 cycles per warp (1.x)
  - sqrt(x) == rcp(rsqrt(x)) == 32 cycles per warp (1.x)

# Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but low accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy for any x
    - Examples: sin(x), exp(x), pow(x,y)

- The -use_fast_math compiler option forces every func() to compile to __func()

# Make your program float-safe!

- Compute 1.3 and beyond have double precision support
  - Double precision will have additional performance cost
  - Careless use of double or undeclared types may run more slowly
- 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`

# Deviations from IEEE-754

- 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
- Combine to IEEE FMA in compute 2.x
- Division
  - 2 ulp error on compute 1.x
  - Fully compliant for 2.x
- Denormalized numbers are supported in compute 2.x and beyond
- No mechanism to detect floating-point exceptions

# Conclusion