CSE 599 I
Accelerated Computing - Programming GPUS

Floating Point Considerations

GPU Teaching Kit

Accelerated Computing

Module 12.1 – Floating-Point Considerations

Floating-Point Precision and Accuracy

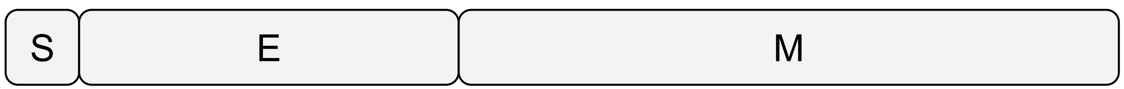
Objective

- To understand the fundamentals of floating-point representation
- To understand the IEEE-754 Floating Point Standard
- CUDA GPU Floating-point speed, accuracy and precision
  - Cause of errors
  - Algorithm considerations
  - Deviations from IEEE-754
  - Accuracy of device runtime functions
  - Future performance considerations

What is IEEE floating-point format?

- An industry-wide standard for representing floating-point numbers to ensure that hardware from different vendors generate results that are consistent with each other

- 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 = $(-1)^S * 1.M * \{2^{E\text{-}bias}\}$

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

Normalized Representation

- Using the definition 1.M as opposed to just M has two advantages
  - One bit is saved, because the initial 1 is implied
    - The remaining part of the mantissa is sometimes referred to as the fraction
  - There is only one representation of (almost) every value
    - For example, the only mantissa value allowed for 0.5_D is M =1.0, with the exponent set to -1, i.e. $0.5_D = 1.0_B * 2^{-1}$
    - Without enforcing the leading one followed by a decimal point, we could have $0.5_D = 0.1_B * 2^0$ or $0.5_D = 10.0_B * 2^{-2}$

Exponent Representation

- In an n-bit 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

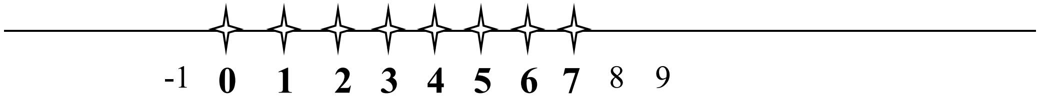
- Assume 1-bit S, 2-bit E, and 2-bit M
  - 0.5D = 1.00_B * 2-1
  - 0.5D = 0 00 00, where S = 0, E = 00, and M = (1.)00

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

Representable Numbers

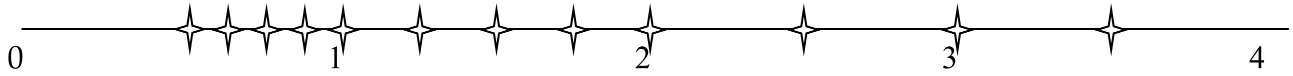
- The representable numbers of a given format is the set of all numbers that can be exactly represented in the format.
- See Table for representable numbers of an unsigned 3-bit integer format

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



No-Zero

– The straightforward implementation
  – However, zero is not a representable number in this format
  – Not acceptable for most any application



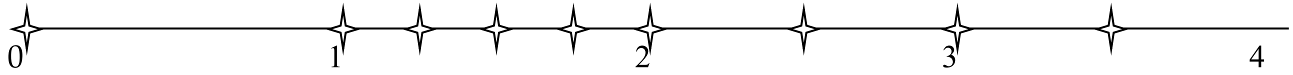
Representable Numbers of ... Format

Cannot Represent Zero!

		No-zero				Gradual underflow	
E	M	S=0	S=1				
00	00	2^{-1}	$-(2^{-1})$				
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$				
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$				
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$				
01	00	2^{0}	$-(2^{0})$				
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$				
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$				
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$				
10	00	2^{1}	$-(2^{1})$				
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$				
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$				
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$				
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 representable numbers will be removed



Flush to Zero

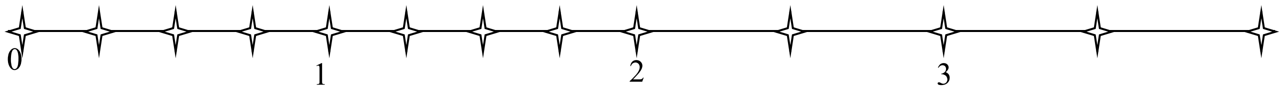
		No-zero		**Flush to Zero**		Denormalized	
E	M	S=0	S=1	**S=0**	**S=1**		
00	00	2^{-1}	$-(2^{-1})$	**0**	**0**		
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	**0**	**0**		
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	**0**	**0**		
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	**0**	**0**		
01	00	2^{0}	$-(2^{0})$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

Why is flushing to zero problematic?

- Many physical model calculations work on values that are very close to zero
  - Dark (but not totally black) sky in movie rendering
  - Small distance fields in electrostatic potential calculation
  - …
- Without Denormalization, these calculations tend to create artifacts that compromise the integrity of the models

Denormalized Numbers

– The actual method adopted by the IEEE standard is called “denormalized 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

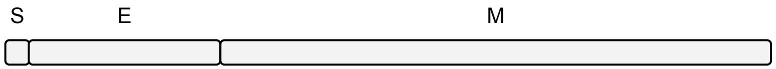


Denormalization

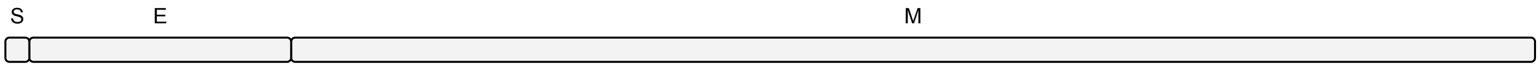
		No-zero		Flush to Zero		**Denormalized**	
E	M	S=0	S=1	S=0	S=1	**S=0**	**S=1**
00	00	2^{-1}	$-(2^{-1})$	0	0	**0**	**0**
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$\mathbf{1*2^{-2}}$	$\mathbf{-1*2^{-2}}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$\mathbf{2*2^{-2}}$	$\mathbf{-2*2^{-2}}$
	11	$2^{-1}+3*2^{-3}$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

IEEE 754 Format and Precision

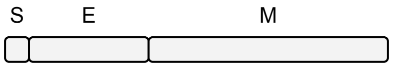
- Single Precision
  - 1-bit sign, 8 bit exponent (bias-127 excess), 23 bit fraction



- Double Precision
  - 1-bit sign, 11-bit exponent (1023-bias excess), 52 bit fraction
  - The largest error for representing a number is reduced to 1/229 of single precision representation



- Half Precision
  - 1-bit sign, 5 bit exponent (bias-15 excess), 10 bit fraction



Special Bit Patterns

exponent	mantissa	meaning
11…1	≠ 0	NaN
11…1	=0	$(-1)^S * \infty$
00…0	≠0	denormalized
00…0	=0	0

- An ∞ can be created by overflow, e.g., divided by zero. Any representable number divided by +∞ or -∞ results in 0.
- NaN (Not a Number) is generated by operations whose input values do not make sense, for example, 0/0, 0*∞, ∞/∞, ∞ - ∞.
  - Also used to for data that has not been properly initialized in a program.
  - Signaling NaNs (SNaNs) are represented with most significant mantissa bit cleared whereas quiet NaNs are represented with most significant mantissa bit set.

Floating Point Accuracy and Rounding

- The accuracy of a floating point arithmetic operation is measured by the maximal error introduced by the operation.
- The most common source of error in floating point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding.
- Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly.

Rounding and Error

– Assume our 5-bit representation, consider

$1.0*2^{-2}$ (0, 00, 01) + $1.00*2^{1}$ (0, 10, 00)

denorm

– The hardware needs to shift the mantissa bits in order to align the correct bits with equal place value

$0.001*2^{1}$ (0, 00, 0001) + $1.00*2^{1}$ (0, 10, 00)

The ideal result would be $1.001 * 2^{1}$ (0, 10, 001) but this would require 3 mantissa bits!

Rounding and Error

- In some cases, the hardware may only perform the operation on a limited number of bits for speed and area cost reasons
  - An adder may only have 3 bit positions in our example so the first operand would be treated as a 0.00

$0.001*2^1$ (0, 00, 0001) + $1.00*2^1$ (0, 10, 00)

Error Measure

- Floating-point operation errors are typically measured using “Units in the Last Place” (ULP)
  - This refers to the place value of the last bit in the mantissa
  - Note that this metric is exponent-dependent

- The best any hardware can do is 0.5 ULP
  - The error is limited by the precision for this case; even if the results were computed to infinite precision, it must be rounded to fit into a fixed-sized representation

- The IEEE standard states that any compliant hardware should compute operations such as multiplication and addition to 0.5 ULP

Order of Operations Matter

- Floating point operations are not strictly associative
- The root cause is that sometimes a very small number can disappear when added to or subtracted from a very large number.
  - (Large + Small) + Small ≠ Large + (Small + Small)

Algorithm Considerations

- Sequential sum

  $1.00*2^{0} + 1.00*2^{0} + 1.00*2^{-2} + 1.00*2^{-2}$

  $= 1.00*2^{1} + 1.00*2^{-2} + 1.00*2^{-2}$

  $= 1.00*2^{1} + 1.00*2^{-2}$

  $= 1.00*2^{1}$

- Parallel reduction

  $(1.00*2^{0} + 1.00*2^{0}) + (1.00*2^{-2} + 1.00*2^{-2})$

  $= 1.00*2^{1} + 1.00*2^{-1}$

  $= 1.0\underline{1}*2^{1}$

atomicAdd + Floating-Point = Stochastic!

- Order of operations matters, i.e. different orderings of additions can yield different results
- When writing from multiple threads to the same memory location, there is a race condition
- atomicAdd ensures that all values get added, but the ordering is still random, based on which thread wins the race
- Therefore, the order of additions is random, which means the result is random!

Make your program float-safe!

- Modern GPU hardware has 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
```

GPU Teaching Kit

Accelerated Computing

Further Reading:

"What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg:

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html