GPU Teaching Kit

Accelerated Computing

Module 12 – Floating-Point Considerations

Lecture 12.1 - 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
  - Accuracy of device runtime functions
  - -fastmath compiler option
  - Future performance considerations

What is IEEE floating-point format?

- An industrywide 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 = $(-1)^S * M * \{2^E\}$, where $1.0 \leq M < 10.0_B$

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

Normalized Representation

- Specifying that $1.0_B \leq M < 10.0_B$ makes the mantissa value for each floating point number unique.
  - For example, the only 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.
  - Mantissa without implied 1 is called the *fraction*

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

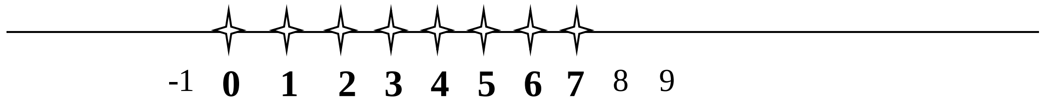
A simple, hypothetical 5-bit FP format

- 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

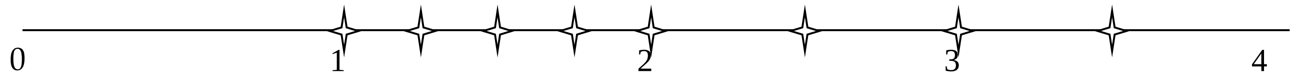
Representable Numbers of … Format

Cannot Represent Zero!

		Non-zero				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 representable numbers will be removed



Flush to Zero

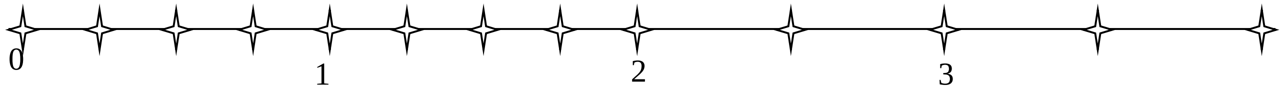
		No-zero		**Flush to Zero**		Denormalized	
E	M	S=0	S=1	**S=0**	**S=1**	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	**0**	**0**	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	**0**	**0**	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	**0**	**0**	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$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

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 * $2^{-2\text{\textasciicircum}(n-1)+2}$



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 $1/2^{29}$ of single precision representation

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)

exponent is 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

- If a hardware adder has at least two more bit positions than the total (both implicit and explicit) number of mantissa bits, the error would never be more than half of the place value of the mantissa
  - 0.001 in our 5-bit format

- We refer to this as 0.5 ULP (Units in the Last Place)
  - If the hardware is designed to perform arithmetic and rounding operations perfectly, the most error that one should introduce should be no more than 0.5 ULP
  - The error is limited by the precision for this case.

Order of Operations Matter

- Floating point operations are not strictly associative
- The root cause is that some times 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

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

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

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

Runtime Math Library

- There are two types of runtime math operations
  - `__func()`: direct mapping to hardware ISA
    - Fast but low accuracy
    - Examples: `__sin(x), __exp(x), __pow(x,y)`
  - `func()` : compile to multiple instructions
    - Slower but higher accuracy (0.5 ulp, units in the least place, or less)
    - 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!

- Double precision likely have performance cost
  - Careless use of double or undeclared types may run more slowly
- Important to 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

GPU Teaching Kit

Accelerated Computing

Module 12 – Floating-Point Considerations

Lecture 12.2 - Numerical Stability

Objective

- Understand numerical stability in linear system solver algorithms
  - Cause of numerical instability
  - Pivoting for increased stability

Numerical Stability

- Linear system solvers may require different ordering of floating-point operations for different input values in order to find a solution
- An algorithm that can always find an appropriate operation order and thus a solution to the problem is a numerically stable algorithm
  - An algorithm that falls short is numerically unstable

Gaussian Elimination Example

Original

$$\begin{aligned} 3X &+ 5Y + 2Z = 19 \\ 2X &+ 3Y + Z = 11 \\ X &+ 2Y + 2Z = 11 \end{aligned} \quad \Rightarrow \quad \begin{aligned} X &+ 5/3Y + 2/3Z = 19/3 \\ X &+ 3/2Y + 1/2Z = 11/2 \\ X &+ 2Y + 2Z = 11 \end{aligned}$$

Step 1: divide equation 1 by 3, equation 2 by 2

$$\Rightarrow \quad \begin{aligned} X &+ 5/3Y + 2/3Z = 19/3 \\ &- 1/6Y - 1/6Z = -5/6 \\ & 1/3Y + 4/3Z = 14/3 \end{aligned}$$

Step 2: subtract equation 1 from equation 2 and equation 3

Gaussian Elimination Example (Cont.)

$$\begin{aligned} X &+ 5/3Y + 2/3Z = 19/3 \\ &- 1/6Y - 1/6Z = -5/6 \\ &\quad 1/3Y + 4/3Z = 14/3 \end{aligned} \Rightarrow \begin{aligned} X &+ 5/3Y + 2/3Z = 19/3 \\ &\quad Y + Z = 5 \\ &\quad Y + 4Z = 14 \end{aligned}$$

Step 3: divide equation 2 by -1/6
and equation 3 by 1/3

$$\Rightarrow \begin{aligned} X &+ 5/3Y + 2/3Z = 19/3 \\ &\quad Y + Z = 5 \\ &\quad + 3Z = 9 \end{aligned}$$

Step 4: subtract equation 2
from equation 3

Gaussian Elimination Example (Cont.)

$$\begin{aligned} X + \tfrac{5}{3}Y + \tfrac{2}{3}Z &= \tfrac{19}{3} \\ Y + Z &= 5 \\ 3Z &= 9 \end{aligned} \quad \Rightarrow \quad \begin{aligned} X + \tfrac{5}{3}Y + \tfrac{2}{3}Z &= \tfrac{19}{3} \\ Y + Z &= 5 \\ Z &= 3 \end{aligned}$$

Step 5: divide equation 3 by 3
We have solution for Z!

$$\Rightarrow \quad \begin{aligned} X + \tfrac{5}{3}Y + \tfrac{2}{3}Z &= \tfrac{19}{3} \\ Y &= 2 \\ Z &= 3 \end{aligned}$$

Step 6: substitute Z solution into equation 2. Solution for Y!

$$\Rightarrow \quad \begin{aligned} X &= 1 \\ Y &= 2 \\ Z &= 3 \end{aligned}$$

Step 7: substitute Y and Z into equation 1. Solution for X!

$$\begin{array}{cccc} 3 & 5 & 2 & 19 \\ 2 & 3 & 1 & 11 \\ 1 & 2 & 2 & 11 \end{array} \rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ 1 & 3/2 & 1/2 & 11/2 \\ 1 & 2 & 2 & 11 \end{array} \rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & -1/6 & -1/6 & -5/6 \\ & 1/3 & 4/3 & 14/3 \end{array}$$

Original

Step 1: divide row 1 by 3, row 2 by 2

Step 2: subtract row 1 from row 2 and row 3

$$\rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 1 & 5 \\ & 1 & 4 & 14 \end{array}$$

Step 3: divide row 2 by -1/6 and row 3 by 1/3

$$\rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 1 & 5 \\ & & 3 & 9 \end{array}$$

Step 4: subtract row 2 from row 3

$$\rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 1 & 5 \\ & & 1 & 3 \end{array}$$

Step 5: divide equation 3 by 3
Solution for Z!

$$\rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & & 2 \\ & & 1 & 3 \end{array}$$

Step 6: substitute Z solution into equation 2. Solution for Y!

$$\rightarrow \begin{array}{cccc} 1 & & & 1 \\ & 1 & & 2 \\ & & 1 & 3 \end{array}$$

Step 7: substitute Y and Z into equation 1. Solution for X!

Basic Gaussian Elimination is Easy to Parallelize

- Have each thread to perform all calculations for a row
  - All divisions in a division step can be done in parallel
  - All subtractions in a subtraction step can be done in parallel
  - Will need barrier synchronization after each step

- However, there is a problem with numerical stability

Pivoting

$$\begin{array}{cccc} & 5 & 2 & 16 \\ 2 & 3 & 1 & 11 \\ 1 & 2 & 2 & 11 \end{array} \quad \rightarrow \quad \begin{array}{cccc} 2 & 3 & 1 & 11 \\ & 5 & 2 & 16 \\ 1 & 2 & 2 & 11 \end{array}$$

Pivoting: Swap row 1 (Equation 1) with row 2 (Equation 2)

$$\rightarrow \quad \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 5 & 2 & 16 \\ 1 & 2 & 2 & 11 \end{array}$$

Step 1: divide row 1 by 3, no need to divide row 2 or row 3

Pivoting (Cont.)

$$\begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 5 & 2 & 16 \\ 1 & 2 & 2 & 11 \end{array} \Rightarrow \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 5 & 2 & 16 \\ & 1/2 & 3/2 & 11/2 \end{array}$$

Step 2: subtract row 1 from row 3
(column 1 of row 2 is already 0)

$$\Rightarrow \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 1 & 2/5 & 16/5 \\ & 1 & 3 & 11 \end{array}$$

Step 3: divide row 2 by 5 and row 3 by 1/2

Pivoting (Cont.)

$$\begin{matrix} 1 & 3/2 & 1/2 & 11/2 \\ & 1 & 2/5 & 16/5 \\ & 1 & 3 & 11 \end{matrix} \Rightarrow \begin{matrix} 1 & 3/2 & 1/2 & 11/2 \\ & 1 & 2/5 & 16/5 \\ & & 13/5 & 39/5 \end{matrix}$$

Step 4: subtract row 2 from row 3

$$\Rightarrow \begin{matrix} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 2/5 & 16/5 \\ & & 1 & 3 \end{matrix}$$

Step 5: divide row 3 by 13/5
Solution for Z!

Pivoting (Cont.)

$$\begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 2/5 & 16/5 \\ & & 1 & 3 \end{array} \Rightarrow \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & & 2 \\ & & 1 & 3 \end{array}$$

Step 6: substitute Z solution into equation 2. Solution for Y!

$$\Rightarrow \begin{array}{cccc} 1 & & & 1 \\ & 1 & & 2 \\ & & 1 & 3 \end{array}$$

Step 7: substitute Y and Z into equation 1. Solution for X!

$$\begin{array}{cccc} & 5 & 2 & 16 \\ 2 & 3 & 1 & 11 \\ 1 & 2 & 2 & 11 \end{array} \;\Rightarrow\; \begin{array}{cccc} 2 & 3 & 1 & 11 \\ & 5 & 2 & 16 \\ 1 & 2 & 2 & 11 \end{array} \;\Rightarrow\; \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 5 & 2 & 16 \\ 1 & 2 & 2 & 11 \end{array}$$

Original

Pivoting: Swap row 1 (Equation 1) with row 2 (Equation 2)

Step 1: divide row 1 by 3, no need to divide row 2 or row 3

$$\Rightarrow\; \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 5 & 2 & 16 \\ & 1/2 & 3/2 & 11/2 \end{array} \;\Rightarrow\; \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 1 & 2/5 & 16/5 \\ & 1 & 3 & 11 \end{array}$$

Step 2: subtract row 1 from row 3 (column 1 of row 2 is already 0)

Step 3: divide row 2 by 5 and row 3 by 1/2

$$\Rightarrow\; \begin{array}{cccc} 1 & 3/2 & 1/2 & 11/2 \\ & 1 & 2/5 & 16/5 \\ & & 13/5 & 39/5 \end{array} \;\Rightarrow\; \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & 2/5 & 16/5 \\ & & 1 & 3 \end{array}$$

Step 4: subtract row 2 from row 3

Step 5: divide row 3 by 13/5
Solution for Z!

$$\Rightarrow\; \begin{array}{cccc} 1 & 5/3 & 2/3 & 19/3 \\ & 1 & & 2 \\ & & 1 & 3 \end{array} \;\Rightarrow\; \begin{array}{cccc} 1 & & & 1 \\ & 1 & & 2 \\ & & 1 & 3 \end{array}$$

Step 6: substitute Z solution into equation 2. Solution for Y!

Figure 7.11

Step 7: substitute Y and Z into equation 1. Solution for X!

Why is Pivoting Hard to Parallelize?

- Need to scan through all rows (in fact columns in general) to find the best pivoting candidate
  - A major disruption to the parallel computation steps
  - Most parallel algorithms avoid full pivoting
  - Thus most parallel algorithms have some level of numerical instability

GPU Teaching Kit

Accelerated Computing