

Lecture 6: Data representation, precision formats and tensor cores

Informatik elective: GPU Computing

Pratik Nayak

Licensed under



In this session

- Data representation and precision formats:
 - IEEE 754: Integral and Floating point types
 - Other available formats.
- Tensor cores:
 - Benefits
 - Functionality
 - Libraries

Bits and bytes

- In C and CUDA, everything can be thought of as a block of 8 bits: “byte”
- Each address (pointer) represents a byte in memory.
- You cannot address a bit, you only address at a byte level

| | | | |
|---------------|------|------|------|
| Values (char) | 23 | 33 | -10 |
| Address(dec) | 200 | 201 | 202 |
| Address(hex) | 0xc8 | 0xc9 | 0xca |

Representing integers

- The set $x \in \mathbb{Z}$
- With CUDA, as in C, you have signed and unsigned representations.
 - unsigned: Only represent non-negative numbers
 - signed: Can represent positive, zero and negative numbers
- With unsigned, you have a one to one correspondence from binary to decimal.
 - Given w bits, you can represent 2^w unsigned integers from $0 \rightarrow 2^w - 1$, which is called its range
 - Example: a 32-bit `int` can represent integers from $0 \rightarrow (2^{32} - 1)$

Bit-widths of Integral types

- Larger the number of bits, larger the maximum representable number.
- In C, the following are common bit width representations of the integral types.
- C-standard specifies only the lower bounds and hence different types of machines (32 bit and 64 bit) **may** have different bit-widths.

| Type | Width (bits) |
|----------------|--------------|
| char | 8 |
| unsigned char | 8 |
| short | 16 |
| unsigned short | 16 |
| int | 32 |
| unsigned int | 32 |
| long | 64 |
| unsigned long | 64 |

Representing signed integers

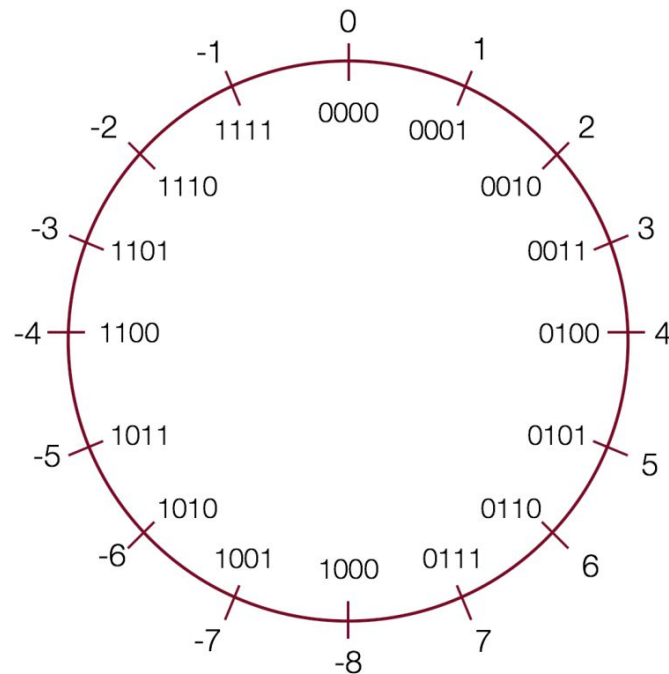
- A little more involved for signed integers. Can use 1 bit to represent sign, but arithmetic gets more complicated (need to borrow and carry, keep track of sign changes etc)
- The canonical way is to use two's complement
 - Negative numbers are obtained by inverting all bits of its positive counterpart (called one's complement) and then adding 1.

For a vector with the bit representation of a w -bit integer $x_{w-1}x_{w-2}\dots x_0$, $x = [x_{w-1}, x_{w-2}, \dots, x_0]$, the binary to two's complement is given by the function:

$$B2T(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

Representing integers

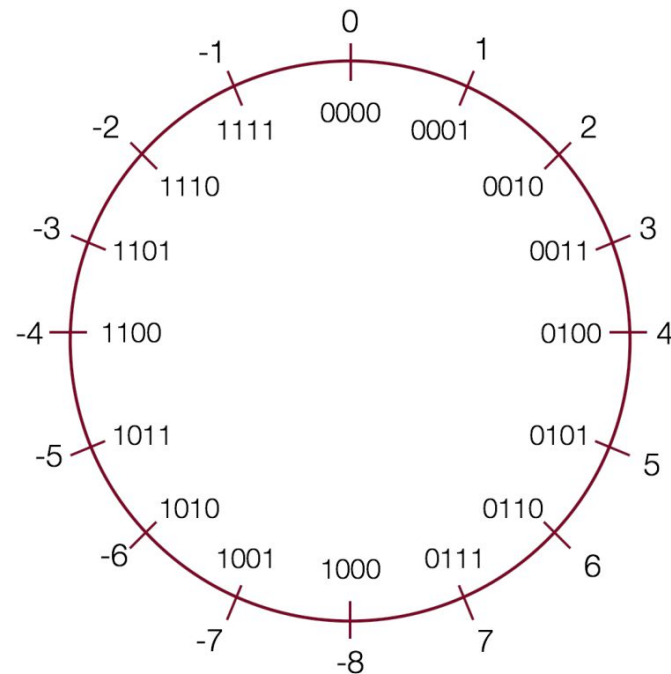
- With this system, you have the following advantages:
 - Only 1 zero.
 - Highest order bit is 1 for negative numbers, 0 for positive numbers
 - Adding two numbers is just bitwise addition
 - Subtraction is a two's complement on one number, and then addition.
 - Multiplication is binary multiplication and then discarding the overflow bits.



Representing integers

- When converting to decimal, the bits representing the power-of-two place still stands:

$$\begin{aligned} -5 &= 1 \quad 0 \quad 1 \quad 1 \\ &= (-1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) \end{aligned}$$



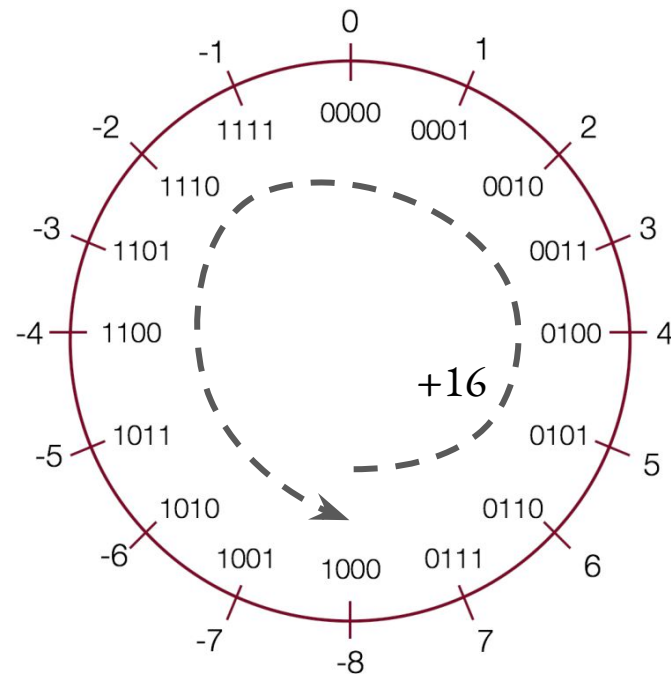
Representing integers

- When converting to decimal, the bits representing the power-of-two place still stands:

$$\begin{aligned}
 -5 &= 1 \quad 0 \quad 1 \quad 1 \\
 &= (-1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0)
 \end{aligned}$$

- Note the maximum representable number:
 - 4 bits \rightarrow -8 to +7: $-2^w \rightarrow (2^w - 1)$
 - Converting -8 (1000) to two's complement gives back -8 (1000). Over the circle and back again \rightarrow

OVERFLOW!



Overflow

- Unsigned integers don't overflow, but just wrap around.
 - For example, trying to represent 65536 with 16 bit `short unsigned int` whose range is $0 \rightarrow (2^{16} - 1)$ will wrap around to 0. ($65535 + 1 = 0$)

Overflow

- Unsigned integers don't overflow, but just wrap around.
 - For example, trying to represent 65536 with 16 bit `short unsigned int` whose range is $0 \rightarrow (2^{16} - 1)$ will wrap around to 0. $(65535 + 1 = 0)$

Sorry, mathematicians!

Overflow

- Unsigned integers don't overflow, but just wrap around.
 - For example, trying to represent 65536 with 16 bit `short unsigned int` whose range is $0 \rightarrow (2^{16} - 1)$ will wrap around to 0. ($65535 + 1 = 0$)

Sorry, mathematicians!
- Signed integers will overflow, and their behaviour according to the C standard is undefined
 - But with most systems (that use Two's complement), it overflows predictably:
 - Example, trying to represent 32768 with 16 bit (2 bytes) `short int` whose range is $-2^8 \rightarrow (2^8 - 1)$ will be -32768 ($32767 + 1 = -32768$)

Overflow

- Unsigned integers don't overflow, but just wrap around.
 - For example, trying to represent 65536 with 16 bit `short unsigned int` whose range is $0 \rightarrow (2^{16} - 1)$ will wrap around to 0. ($65535 + 1 = 0$)

Sorry, mathematicians!
- Signed integers will overflow, and their behaviour according to the C standard is undefined
 - But with most systems (that use Two's complement), it overflows predictably:
 - Example, trying to represent 32768 with 16 bit (2 bytes) `short int` whose range is $-2^8 \rightarrow (2^8 - 1)$ will be -32768 ($32767 + 1 = -32768$)

You always need to look at the number ranges you want to represent and select bit-widths appropriately.

Addressing and byte ordering

- Given 32 bit machines, we are able to address $2^{32} \sim 4GB$, because each byte is individually addressable by a program.
- With 64 bit machines, the number of addressable bytes are $2^{64} \sim 16EB \sim 16 \times 10^9GB$, which is far more than what we will probably ever need.

Byte ordering: Little endian and Big endian

- Given a number, say of `int` type (4 bytes), how do we store it in memory ?
 - We can for example store them as 8 digit hex numbers `0x01234567` and view them

as 0x 01 23 45 67

 0000 0001 0010 0011 0100 0101 0110 0111

 ----- ----- ----- -----

 Most significant → Least significant

- Little endian: If the bytes are ordered from least significant to most significant (the “little-end” comes first.
- Big Endian: If the bytes are ordered from most significant to least significant (the “big-end” comes first.

Representing real numbers

- Real numbers are trickier to represent on computers. Generally classified into 3 types:
 - Rational (type 1): Exactly representable: $\frac{1}{4} = 0.25$
 - Rational (type 2): Not exactly representable (with digit notation): $\frac{1}{3} = 0.333\dots$
 - Irrational numbers: Not exactly representable (with digit notation): $\pi = 3.1415\dots$
- Computation on hardware is done (when non-symbolic or when non-exact unlike in Julia) with digits, therefore, there is a need to represent real numbers similar to what we have for integers.

Representing real numbers


- The challenges:
 - We have a fixed number of bits available.
 - Means that we almost always have errors. Even the rational numbers are not exactly representable.
 - Maximize representable range.
 - Perform calculations efficiently.

Representing real numbers: Fixed-point

- Use the same idea as in integers, with an additional decimal point to differentiate.

- 123.45 ($d_2d_1d_0.d_{-1}d_{-2}$) in decimal would be

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

- What is the representable range ? $0 \rightarrow 999.99$
- What is the precision ? 5 decimal digits 
- Need to **round/truncate** for some numbers: 123.456 , 123.333...
- **Overflow** for numbers beyond max representable: 1000, 999.991, 999.9901 etc
- **Underflow** for small numbers below representable precision: 0.001, 0.002 etc

Representing real numbers: Floating-point

- Fixed-point arithmetic is simple, but it has limitations for its range.
- A more flexible approach is to represent numbers using 3 parts :

$$V = (-1)^s \times M \times 2^E$$

Sign (s): $s=1$ (negative), $s=0$ (positive)

Mantissa (M): Or Significand, the fractional binary number with range: $[1 \rightarrow (2-\epsilon)]$ or $[0 \rightarrow (1-\epsilon)]$

Exponent (E): Weights the value by a power of 2.

- Also known as the IEEE Standard 754 or IEEE Floating point.

Representing real numbers: Floating-point

$$V = (-1)^s \times M \times 2^E$$

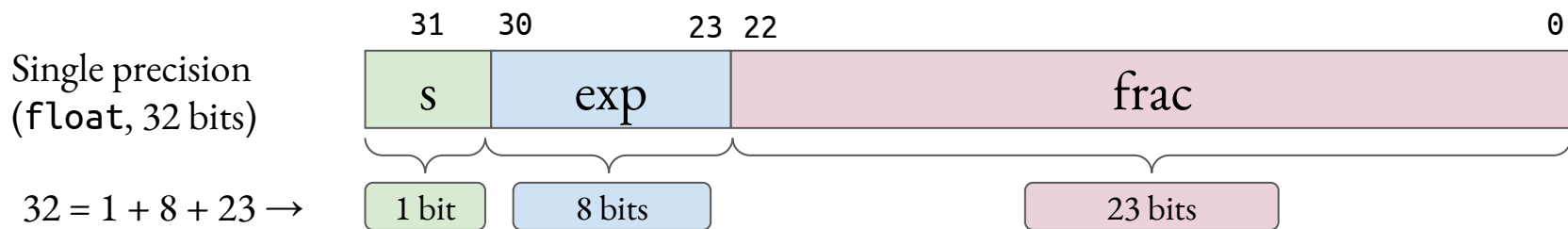
- Example: With $s=0$, $M=1.5$, $E=9$, we have $V = (-1)^0 \times 1.5 \times 2^9 = 768$

With $s=1$, $M=1.5$, $E=9$, we have $V = (-1)^1 \times 1.5 \times 2^9 = -768$

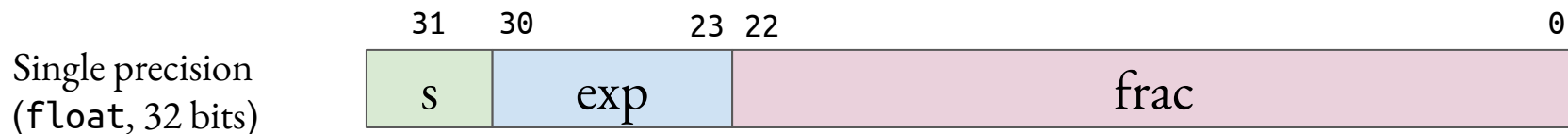
Representing real numbers: Floating-point

$$V = (-1)^s \times M \times 2^E$$

- Bit-representation is also divided into three fields:
 - The single sign bit s , encodes the sign
 - A k -bit exponent field, $exp = e_{k-1} \dots e_1 e_0$ encodes the exponent
 - A n -bit fractional field, $frac = f_{n-1} \dots f_1 f_0$ encodes the mantissa M , with modifications if the exponent field equals 0.

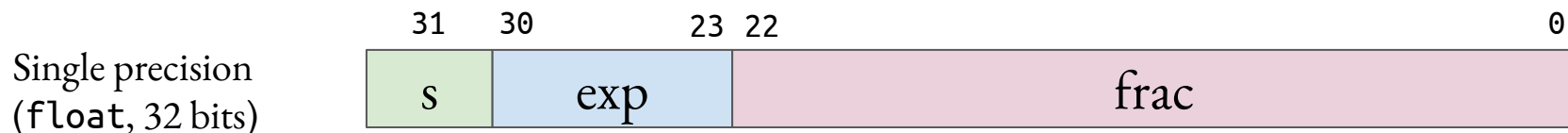


Floating-point: Some internal details



- “Normal float”: If the exponent is not all zeros/ones, that is bits 23→30 don’t represent 0 or 255
- Exponent is a “biased” signed integer: $\text{Exponent} = \text{exp} - \text{bias}$, with $\text{bias} = (2^{k-1} - 1)$, where k is the number of bits in exp. For float, bias is $(2^{8-1} - 1 = 127)$, therefore exp range is $-126 \rightarrow +127$
- Fraction represents the fraction value: $0 \leq f < 1$, with binary representation: $(0.f_{n-1} \dots f_1 f_0)$
- The Mantissa then is defined to be $M = 1 + f$. The implied leading 1 enables getting one digit for free.

Floating-point: Some internal details

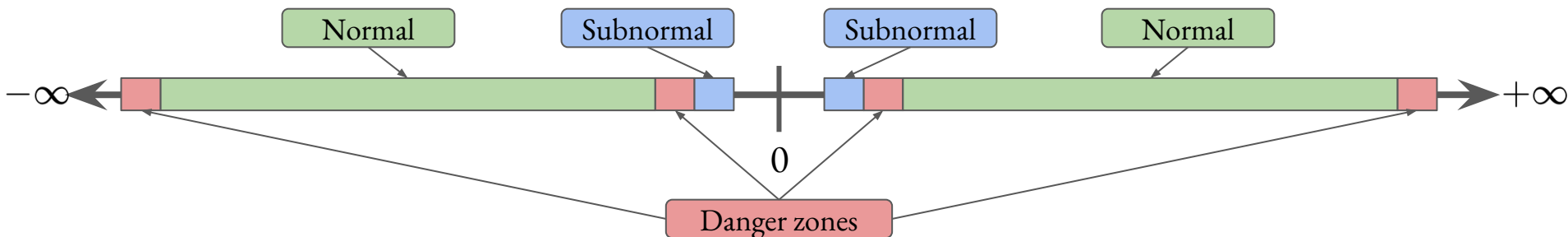


- “Normal float”: If the exponent is not all zeros/ones, that is bits 23→30 don’t represent 0 or 255
- Exponent is a “biased” signed integer: $\text{Exponent} = \text{exp} - \text{bias}$, with $\text{bias} = (2^{k-1} - 1)$, where k is the number of bits in exp. For float, bias is $(2^{8-1} - 1 = 127)$, therefore exp range is $-126 \rightarrow +127$
- Fraction represents the fraction value: $0 \leq f < 1$, with binary representation: $(0.f_{n-1} \dots f_1 f_0)$
- The

The exponent is adjusted so that the mantissa is in the range $1 \leq M < 2$

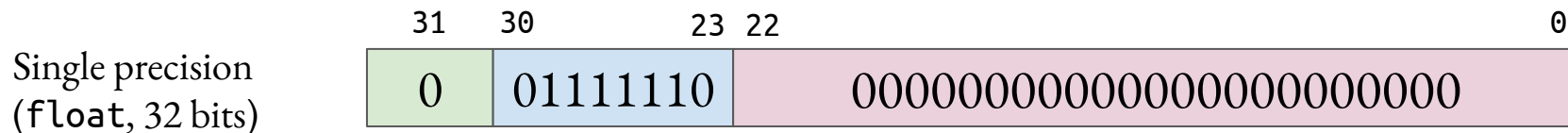
t for free.

Floating-point: Ranges and subnormals



- Can we represent numbers below the smallest possible normal number: $2^{-(E_b-1)}$; $E_b = 2^k - 1$
- Yes, if we fix the exponent to $-(E_b - 1)$, example $-(127 - 1)$ for **float**, then we can use the additional bits for the fraction. Therefore, the smallest representable number becomes:
 - In general, $2^{-(E_b-1)-n}$. For example for **float**: $2^{-126-23} = 2^{-149} \approx 1.4 \times 10^{-45}$
 - Can you similarly figure out the largest representable subnormal? self-study exercise
- These are called Subnormals or Denormals

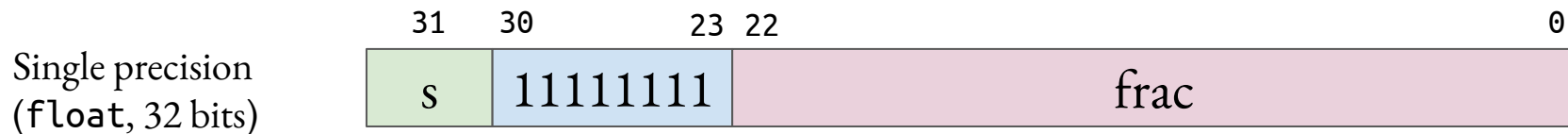
Floating-point: Example



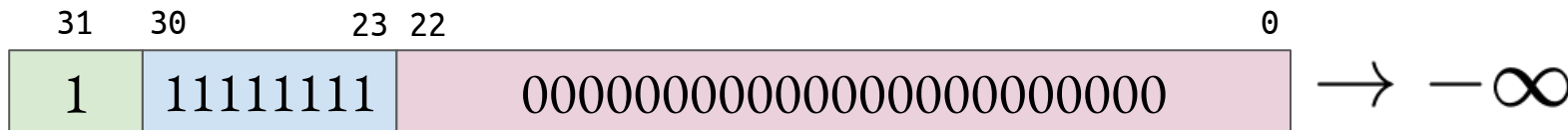
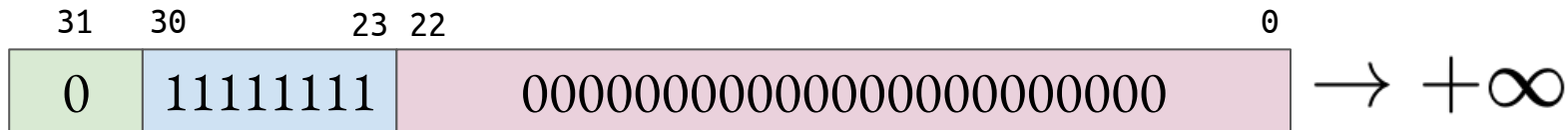
- What value does the bit representation above represent ?
 - Break it down:
 - Sign bit: 0 \rightarrow positive number
 - Exponent: 01111110 = 126, as it is biased, exponent is $126 - 127 = -1$
 - Fraction: 0, therefore Mantissa is 1.0 (binary), which is 1.0 in decimal
 - Therefore, using $V = (-1)^s \times M \times 2^E$, we get $V = (-1)^0 \times 1.0 \times 2^{-1} = 0.5$

Verify if you have understood: <https://www.h-schmidt.net/FloatConverter/IEEE754de.html>

Floating-point: Exceptionals



- “Exceptional float”: If the exponent is not all ones:
- Used to denote infinities and NaNs:



- Divisions : $0/0$, $\pm\infty/\pm\infty$, multiplications $0 \times \pm\infty$ etc produce exceptionals.

Floating-point: Non-associativity

```
#include <iomanip>
#include <iostream>

int main()
{
    float a = 3.14;
    float b = 1e20;

    std::cout << std::setprecision(16)
              << "order 1 ((3.14 + 1e20) - 1e20) : " << ((a + b) - b) << "\n";
    std::cout << std::setprecision(16)
              << "order 2 (3.14 + (1e20 - 1e20)) : " << (a + (b - b)) << "\n";
    return 0;
}
```

```
$ ./fp_assoc_test
order 1 ((3.14 + 1e20) - 1e20) : 0
order 2 (3.14 + (1e20 - 1e20)) : 3.140000104904175
```

Floating-point: Non-associativity

```
#include <iomanip>
#include <iostream>

int main()
{
    float a = 3.14;
    float b = 1e20;

    std::cout << std::setprecision(16)
              << "order 1 ((3.14 + 1e20) - 1e20) : " << ((a + b) - b) << "\n";
    std::cout << std::setprecision(16)
              << "order 2 (3.14 + (1e20 - 1e20)) : " << (a + (b - b)) << "\n";
    return 0;
}
```

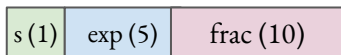
Associativity of operations matters. Order 1 != Order 2

Recommended reference (Goldberg, 1991, ACM CSUR):

<https://dl.acm.org/doi/10.1145/103162.103163>

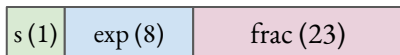
IEEE 754 floating point formats

Half
(half)
(16 bits)



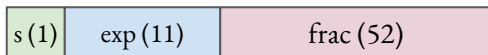
| Bit-width | config (M,E) | roundoff | Normal Range |
|-----------|--------------|---------------------------------------|--|
| 16 | (10,5) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 6.10 \times 10^{-5} \rightarrow 6.55 \times 10^4$ |

Single
(float)
(32 bits)



| | | | |
|----|--------|---------------------------------------|--|
| 32 | (23,8) | $2^{-24} \approx 5.96 \times 10^{-8}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|--------|---------------------------------------|--|

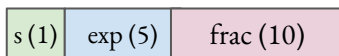
Double
(double)
(64 bits)



| | | | |
|----|---------|--|--|
| 64 | (52,11) | $2^{-53} \approx 1.11 \times 10^{-16}$ | $\pm 2.22 \times 10^{-308} \rightarrow 1.80 \times 10^{308}$ |
|----|---------|--|--|

IEEE 754 floating point formats

Half
(half)
(16 bits)



| Bit-width | config (M,E) | roundoff | Normal Range |
|-----------|--------------|---------------------------------------|--|
| 16 | (10,5) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 6.10 \times 10^{-5} \rightarrow 6.55 \times 10^4$ |

Machine Epsilon/Unit round-off (ϵ):

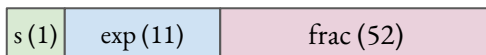
Largest relative rounding error with
round-to-nearest

(23,8)

$2^{-24} \approx 5.96 \times 10^{-8}$

$\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$

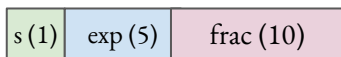
Double
(double)
(64 bits)



| | | | |
|----|---------|--|--|
| 64 | (52,11) | $2^{-53} \approx 1.11 \times 10^{-16}$ | $\pm 2.22 \times 10^{-308} \rightarrow 1.80 \times 10^{308}$ |
|----|---------|--|--|

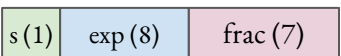
Additional floating point formats

half,
(16 bits)

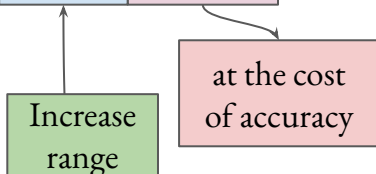


| Bit-width | config (M,E) | roundoff | Normal Range |
|-----------|--------------|---------------------------------------|--|
| 16 | (10,5) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 6.10 \times 10^{-5} \rightarrow 6.55 \times 10^4$ |

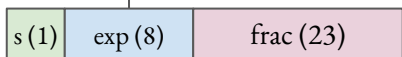
bfloat,
(16 bits)



| | | | |
|----|-------|--------------------------------------|--|
| 16 | (7,8) | $2^{-8} \approx 3.90 \times 10^{-3}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|-------|--------------------------------------|--|

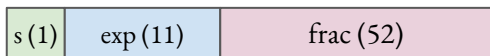


float,
(32 bits)



| | | | |
|----|--------|---------------------------------------|--|
| 32 | (23,8) | $2^{-24} \approx 5.96 \times 10^{-8}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|--------|---------------------------------------|--|

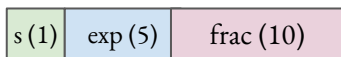
double,
(64 bits)



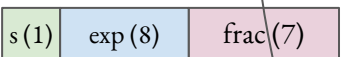
| | | | |
|----|---------|--|--|
| 64 | (52,11) | $2^{-53} \approx 1.11 \times 10^{-16}$ | $\pm 2.22 \times 10^{-308} \rightarrow 1.80 \times 10^{308}$ |
|----|---------|--|--|

Additional floating point formats

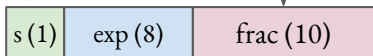
half,
(16 bits)



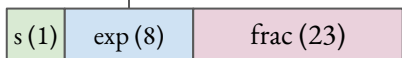
bfloat,
(16 bits)



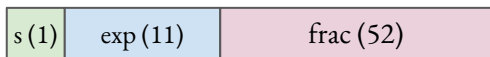
tensorfloat,
(19 bits)



float,
(32 bits)



double,
(64 bits)



| Bit-width | config (M,E) | roundoff | Normal Range |
|-----------|--------------|---------------------------------------|--|
| 16 | (10,5) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 6.10 \times 10^{-5} \rightarrow 6.55 \times 10^4$ |

| | | | |
|----|-------|--------------------------------------|--|
| 16 | (7,8) | $2^{-8} \approx 3.90 \times 10^{-3}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|-------|--------------------------------------|--|

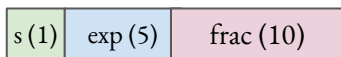
| | | | |
|----|--------|---------------------------------------|--|
| 19 | (10,8) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|--------|---------------------------------------|--|

| | | | |
|----|--------|---------------------------------------|--|
| 32 | (23,8) | $2^{-24} \approx 5.96 \times 10^{-8}$ | $\pm 1.18 \times 10^{-38} \rightarrow 3.40 \times 10^{38}$ |
|----|--------|---------------------------------------|--|

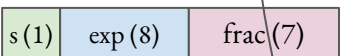
| | | | |
|----|---------|--|--|
| 64 | (52,11) | $2^{-53} \approx 1.11 \times 10^{-16}$ | $\pm 2.22 \times 10^{-308} \rightarrow 1.80 \times 10^{308}$ |
|----|---------|--|--|

Additional floating point formats

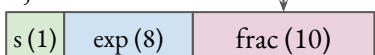
half,
(16 bits)



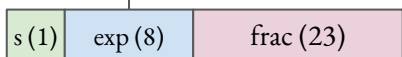
bfloat,
(16 bits)



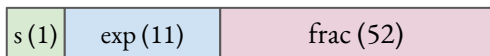
tensorfloat,
(19 bits)



float,
(32 bits)



double,
(64 bits)



| Bit-width | config (M,E) | roundoff | Normal Range |
|-----------|--------------|---------------------------------------|--|
| 16 | (10,5) | $2^{-11} \approx 4.88 \times 10^{-4}$ | $\pm 6.10 \times 10^{-5} \rightarrow 6.55 \times 10^4$ |

- Not really a storage format, but a computation format.
- In the NVIDIA TF32 tensor cores, inputs and output are in FP32, but the high-throughput tensor multiplication is done in TF32

| | | | |
|----|---------|--|--|
| 64 | (52,11) | $2^{-53} \approx 1.11 \times 10^{-16}$ | $\pm 2.22 \times 10^{-308} \rightarrow 1.80 \times 10^{308}$ |
|----|---------|--|--|

Challenges with parallel computing

- If floating point operations are not associative, ensuring deterministic result can be difficult, particularly in the case of hundreds of threads like in CUDA.
- Parallel reductions are an example. See dot product.
- Fixed ordering usually gives a reproducible result, but can kill performance.
- But whether a deterministic result is necessary is an important question.
 - For most applications, it is not and there you benefit from the performance.
- Also non-deterministic nature can lead to lucky rounding (minimizing the accumulated errors) and sometimes prevent stagnation errors (can be an issue in lower precision formats due to smaller range).

Challenges with parallel computing



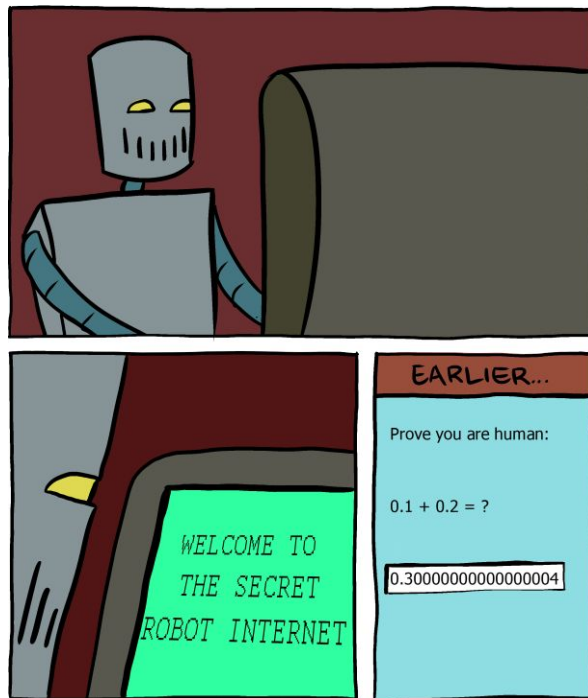
Autopsy Report:

Dr. Andrew Esty

Time of Death: 03/16 11:53

Cause of Death: Rounding Errors

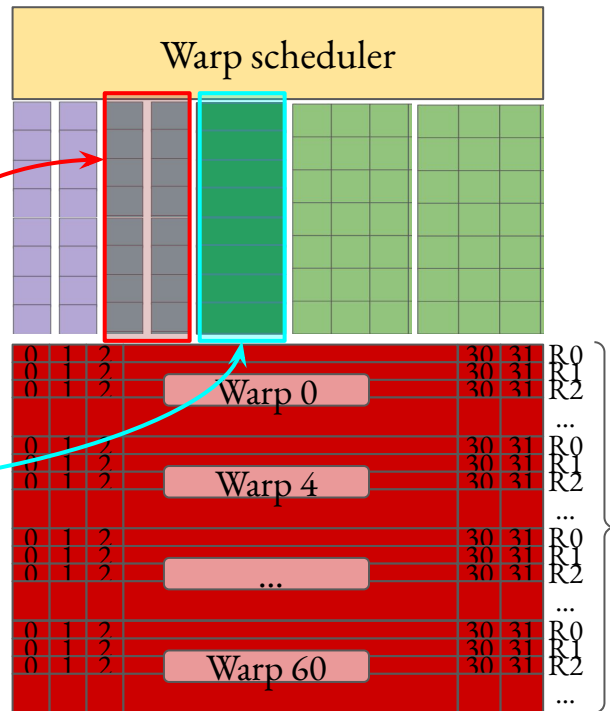
[SMBC comics: id-1118]



[SMBC comics: id-2999]

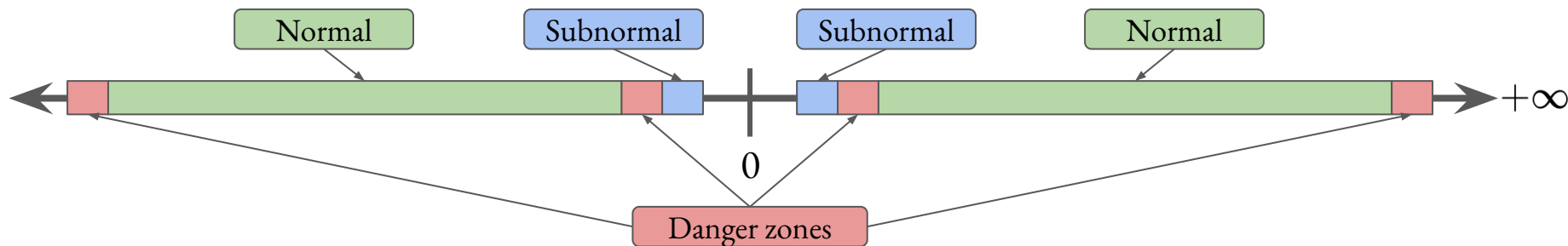
Why lower precision ?

- If possible, always try to use lower precisions, especially on GPUs
 - Lesser data transfer → Higher effective throughput.
 - (4 bytes for float vs 8 bytes for double): A factor of 2 reduction.
 - Generally more parallel compute units for lower precision variants.
 - Example A100: 16 FU for FP32, and only 8 for FP64.
 - Also lesser register usage, lesser shared memory usage → higher overall occupancy.



Why not lower precision everywhere ?

- All algorithms are not all precision-stable.
 - Small variations can cause large instabilities.
- Not all numbers are representable with lower precisions → Underflow, overflow



Possible solutions and ideas

- Scaling: A popular solution to scale all values by largest number to reduce the range that needs to be represented.
- Template your code to adapt to different precisions needed, enabling easy switch when necessary.
- Compression: Lossless compression and lossy compression are both possible with CUDA.
 - SZ, ZFP, FZ are few available compressors.
- Adapt your code to use mixed-precision computations:
 - Use lower-precision where possible, but use higher-precision where necessary: Ensures stability while not not reducing performance.

Tensor cores on CUDA

- Introduced with Volta architecture for high-throughput computation.
- Useful to accelerate matrix-matrix multiplications, a common operation in deep learning and the motivation to add such units to GPUs.
- Tensor cores are units that are able to perform a matrix-multiply-accumulate (MMA) operation
- Note that they are mixed-precision units: The input and output precisions can be different

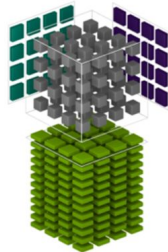
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

HMMA FP16 or FP32
 IMMA INT32

FP16
 INT8 or UINT8

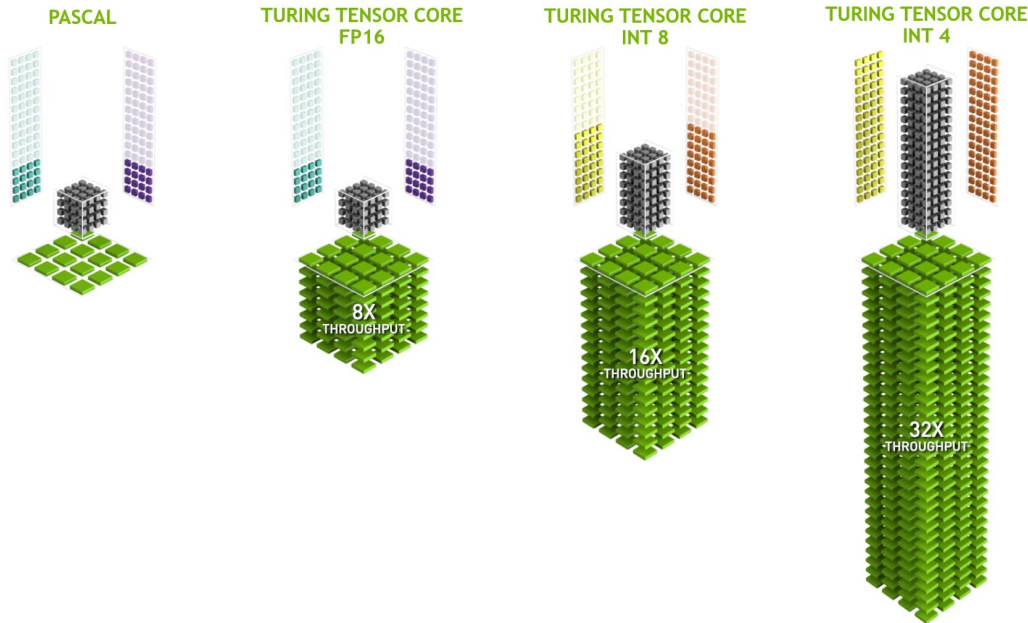
FP16
 INT8 or UINT8

FP16 or FP32
 INT32



Tensor cores on CUDA

- Lower precisions can have significantly higher throughput.



Matrix-matrix multiply with Tensor cores

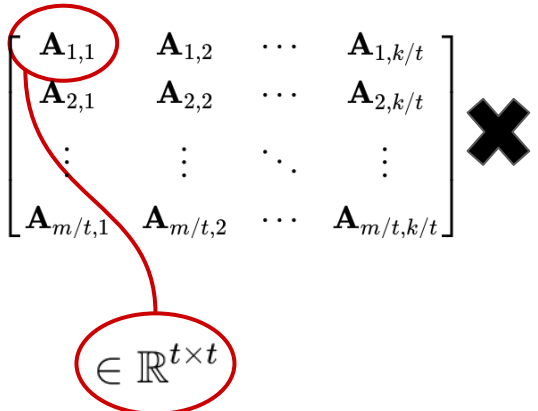
- Consider the operation: $D = A * B + C$, with $D \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$
- Each matrix is divided into tiles of size $t \times t$, the operation can be written as

$$\begin{bmatrix} \mathbf{D}_{1,1} & \mathbf{D}_{1,2} & \cdots & \mathbf{D}_{1,n/t} \\ \mathbf{D}_{2,1} & \mathbf{D}_{2,2} & \cdots & \mathbf{D}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}_{m/t,1} & \mathbf{D}_{m/t,2} & \cdots & \mathbf{D}_{m/t,n/t} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \cdots & \mathbf{A}_{1,k/t} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \cdots & \mathbf{A}_{2,k/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{m/t,1} & \mathbf{A}_{m/t,2} & \cdots & \mathbf{A}_{m/t,k/t} \end{bmatrix} \otimes \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \cdots & \mathbf{B}_{1,n/t} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \cdots & \mathbf{B}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{k/t,1} & \mathbf{B}_{k/t,2} & \cdots & \mathbf{B}_{k/t,n/t} \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \cdots & \mathbf{C}_{1,n/t} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \cdots & \mathbf{C}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{m/t,1} & \mathbf{C}_{m/t,2} & \cdots & \mathbf{C}_{m/t,n/t} \end{bmatrix}$$

Matrix-matrix multiply with Tensor cores

- Consider the operation: $D = A * B + C$, with $D \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$
- Each matrix is divided into tiles of size $t \times t$, the operation can be written as

$$\begin{bmatrix} \mathbf{D}_{1,1} & \mathbf{D}_{1,2} & \cdots & \mathbf{D}_{1,n/t} \\ \mathbf{D}_{2,1} & \mathbf{D}_{2,2} & \cdots & \mathbf{D}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}_{m/t,1} & \mathbf{D}_{m/t,2} & \cdots & \mathbf{D}_{m/t,n/t} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \cdots & \mathbf{A}_{1,k/t} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \cdots & \mathbf{A}_{2,k/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{m/t,1} & \mathbf{A}_{m/t,2} & \cdots & \mathbf{A}_{m/t,k/t} \end{bmatrix} \otimes \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \cdots & \mathbf{B}_{1,n/t} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \cdots & \mathbf{B}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{k/t,1} & \mathbf{B}_{k/t,2} & \cdots & \mathbf{B}_{k/t,n/t} \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \cdots & \mathbf{C}_{1,n/t} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \cdots & \mathbf{C}_{2,n/t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{m/t,1} & \mathbf{C}_{m/t,2} & \cdots & \mathbf{C}_{m/t,n/t} \end{bmatrix}$$



Warp matrix functions

- CUDA provides warp matrix operations that enable you to use the tensor core units, to do those tile $t \times t$ based MMA operations. All threads in the warp must execute the same operation.

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class
fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
fragment<...> &c, bool satf=false);
```

- Main operation is the `mma_sync` call with matrices stored as `fragments`.

Example: MMA using warp matrix functions

```
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    // Initialize the output to zero
    wmma::fill_fragment(c_frag, 0.0f);

    // Load the inputs
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);

    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Store the output
    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

Example: MMA using warp matrix functions

```
#include <mma.h>
using namespace nvcuda;
```

```
__global__ void wmma_ker(half *a, half *b, float *c) {
```

```
    // Declare the fragments
```

```
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
```

```
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
```

```
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
```

```
    // Initialize the output to zero
```

```
    wmma::fill_fragment(c_frag, 0.0f);
```

```
    // Load the inputs
```

```
    wmma::load_matrix_sync(a_frag, a, 16);
```

```
    wmma::load_matrix_sync(b_frag, b, 16);
```

```
    // Perform the matrix multiplication
```

```
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
```

```
    // Store the output
```

```
    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
```

```
}
```

Performs operation: $D = A * B + C$
with 16 matrices (tiles) of size, 16×16

Each fragment's **Use** is noted:
`matrix_a`, `matrix_b` or `accumulator`.

Note that for better accesses, layouts of the fragments can be different

A warp-wide load and sync-barrier

Perform the MMA operation

Store the result to main memory

Libraries

- As you can imagine, this is not simple to get right.
- CUDA provides libraries that do the heavy lifting for you.
 - cuBLAS: If you are using the BLAS functionality and just need the GEMMs
 - cuDNN: If you are doing DNN, then cuDNN you can configure your neural networks for mixed-precision and to use these highly-efficient matrix-matrix multiply functionality.
 - CUTLASS: For the more C++ oriented, a high-level BLAS-type library that you can use in your code, which gives you better abstractions and also has really good mixed-precision support.

Summary

- Floating point formats and arithmetic:
 - Memory representation, usage, hardware support
- Tensor cores: High throughput compute engines
 - Warp MMA functions

Next lecture

- Dynamic parallelism in CUDA
- Asynchronous/Synchronization-free programming in CUDA
 - Asynchronous API
 - Locks, atomics