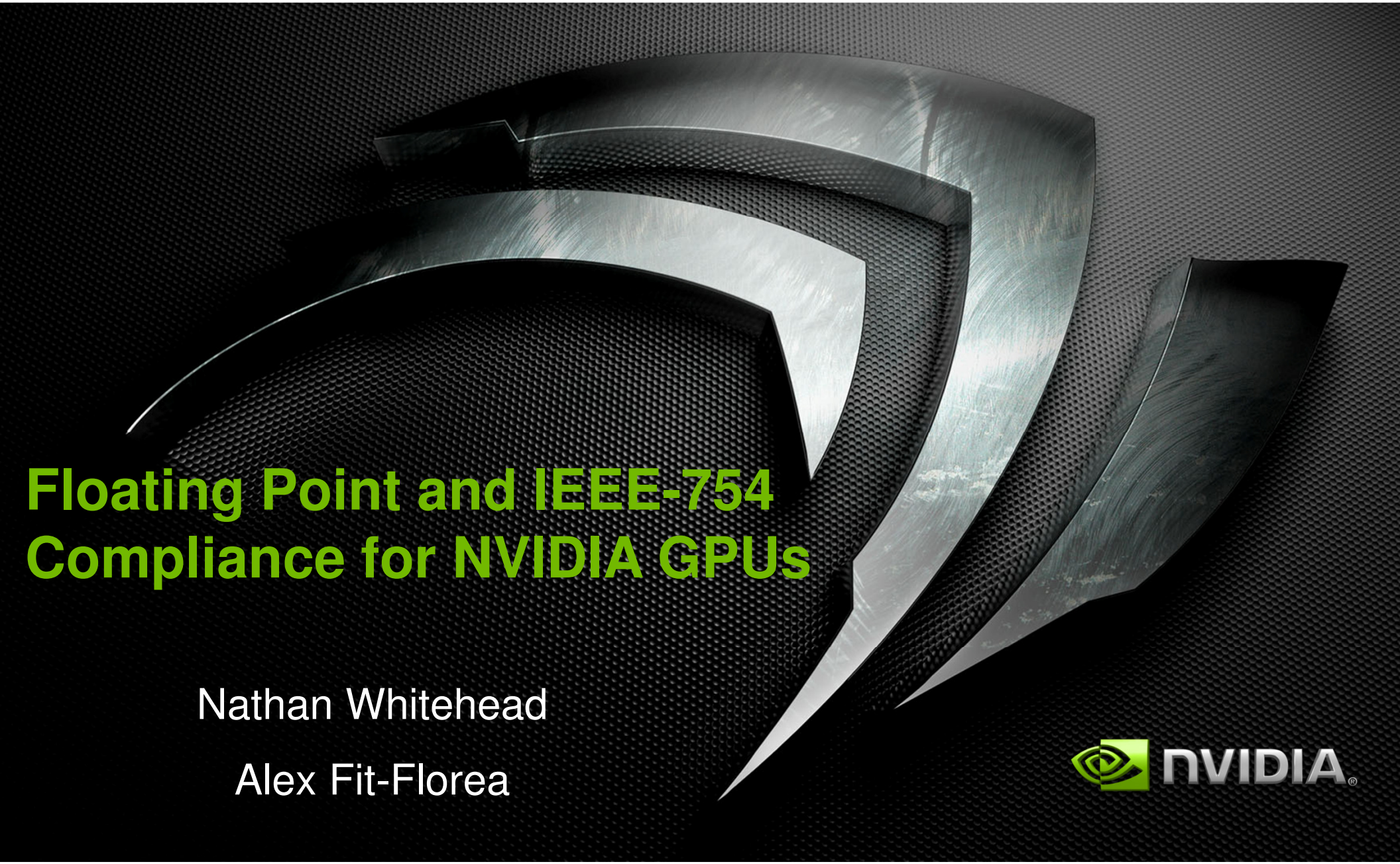

Floating Point and IEEE-754
Compliance for NVIDIA GPUs
Nathan Whitehead
Alex Fit-Florea
NVIDIA

Outline

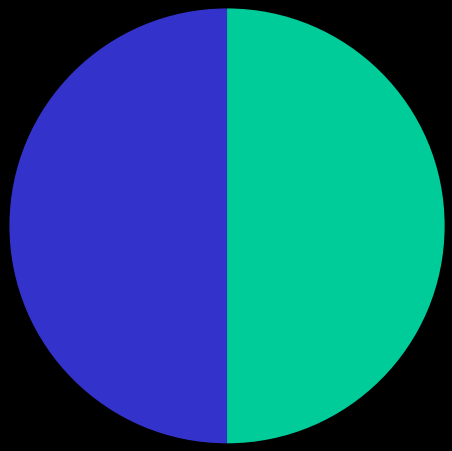
(1) An overview of floating point and IEEE 754

(2) Issues related to accuracy: examples

(3) CUDA, SSE, x87

Floating Point – a first challenge

one half



What is two thirds?

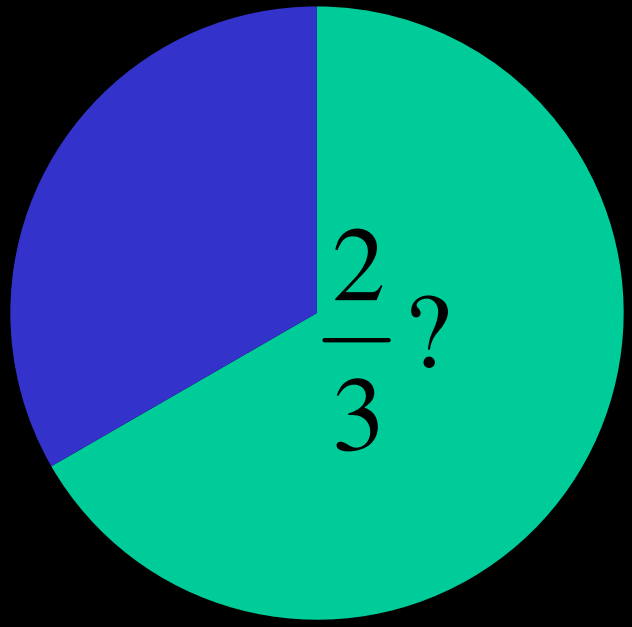
Radix ten: 0.66666…

Scientific: 6.67×10^{-1}

Radix two: 0.101010…

+0.10101010…

$+1 \times 2^{-1} \times +1.01010101\ldots$



+	-1	1.0101010101…011

FP & IEEE 754

sign significand or fraction

exponent (biased)

Value of binary representation

$$(-1)^{s} \times 2^{e-\text{bias}} \times 1.f$$

FP & IEEE 754

Most popular: 32-bit and 64-bit, i.e. single and double precision

	Single precision (SP) a.k.a. **float**	Double precision (DP) a.k.a. **double**
s size	1 bit	1 bit
e size	8 bit	11 bit
bias	+127 = 2^7-1	+1023 = 2^{10} - 1
f size	23 bit	52 bit
range	~ (-3.4e38, 3.4e38)	~ (-1.79e308, 1.79e308)

IEEE 754 Basic Arithmetic

Guarantees unique results for:

(a + b), (a - b)

(a × b), (a + b × c)

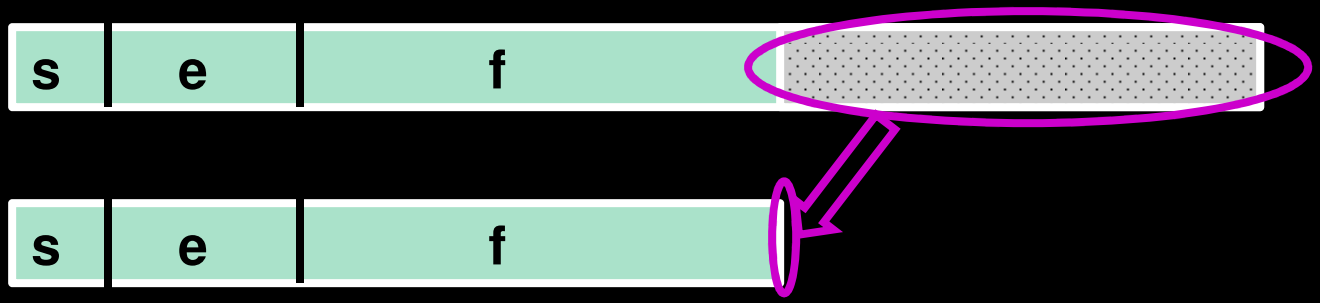
(a / b), sqrt(a)

But not for other frequently used functions lik

exp(a)

cos(a)

Relatively easy to test the supported functions for correct rounding; e.g. (a × b):



IEEE 754 Accuracy over Multiple Operations

A simple sum series

$$S = 1.0 + \sum_{i=0}^{i<N} \frac{1}{N}$$

$$S = 1.0 + N \times \frac{1}{N}$$

$$S = 1.0 + 1$$

$$S = 2.0$$

IEEE 754 Accuracy over Multiple Operations

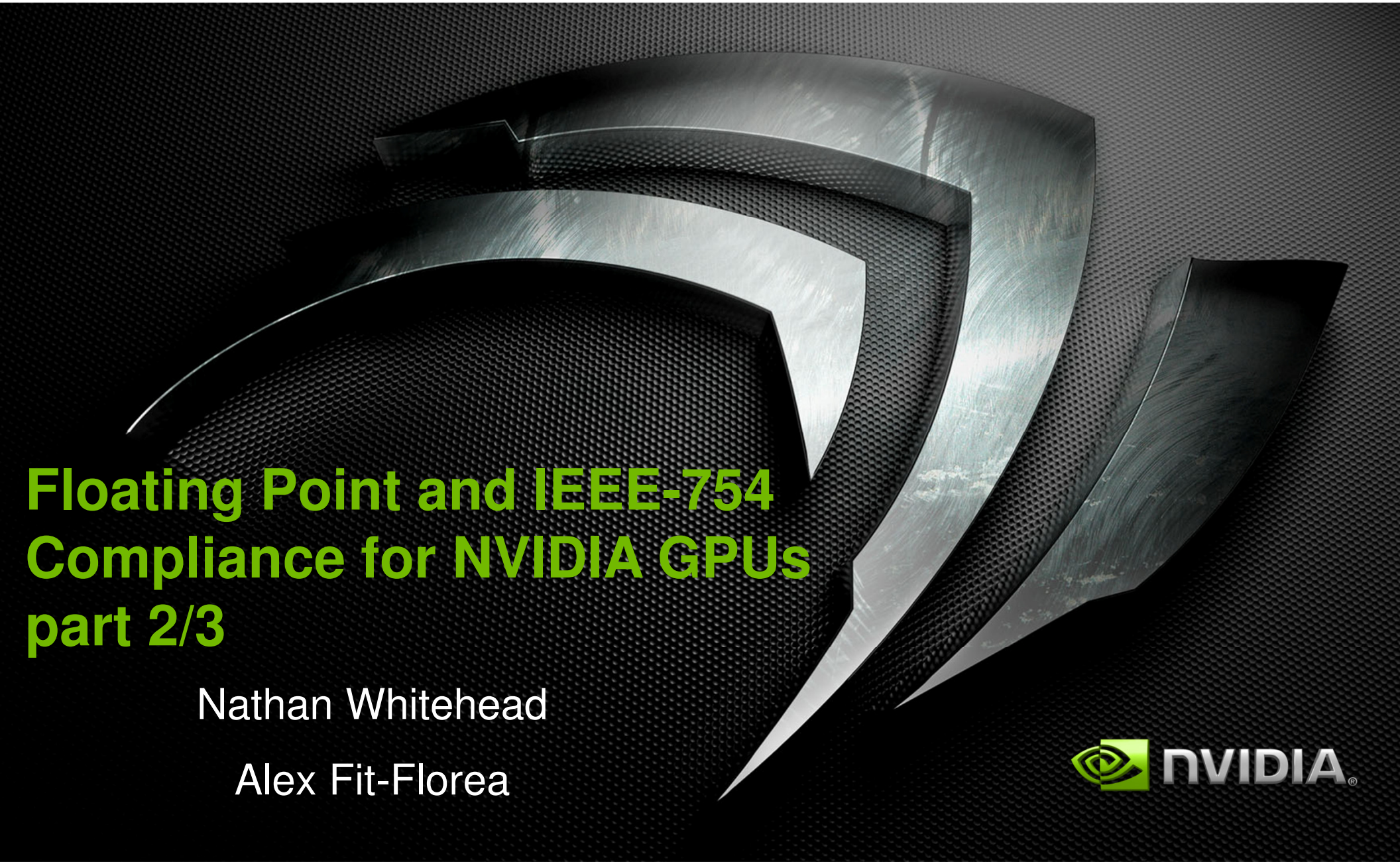
```
#include <stdio.h>
int main() {
   float e = 1.0f / 1000000;
   float s = 1.0f;
   int i;
   for(i = 0; i < 1000000; ++i){
      s += e;
   }
   printf("s = %18.18f\n", s);
}

gcc sum.c -m64 && ./a.out

s = 1.953674316406250000
```

```
#include <stdio.h>
int main() {
   double e = 1.0 / 1000000;
   double s = 1.0;
   int i;
   for(i = 0; i < 1000000; ++i){
      s += e;
   }
   printf("s = %18.18f\n", s);
}

gcc sum.c -m64 && ./a.out

s = 1.999999999917733362
```

IEEE 754 How Execution Order Affects Results

SP example: ((A + B) + C) vs. (A + (B + C))

A	=	$2^{1} \times 1.0000\ 0000\ 0000\ 0000\ 0000\ 001$	
B	=	$2^{0} \times 1.0000\ 0000\ 0000\ 0000\ 0000\ 001$	
C	=	$2^{3} \times 1.0000\ 0000\ 0000\ 0000\ 0000\ 001$	
A + B	=	$2^{1} \times 1.1000\ 0000\ 0000\ 0000\ 0000\ 001$	10000…
RN(A + B)	=	$2^{1} \times 1.1000\ 0000\ 0000\ 0000\ 0000\ 010$	
B + C	=	$2^{3} \times 1.0010\ 0000\ 0000\ 0000\ 0000\ 001$	00100…
RN(B + C)	=	$2^{3} \times 1.0010\ 0000\ 0000\ 0000\ 0000\ 001$	
A + B + C	=	$2^{3} \times 1.0110\ 0000\ 0000\ 0000\ 0000\ 001$	01100…
RN(RN(A + B) + C)	=	$2^{3} \times 1.0110\ 0000\ 0000\ 0000\ 0000\ 010$	
RN(A + RN(B + C))	=	$2^{3} \times 1.0110\ 0000\ 0000\ 0000\ 0000\ 001$	



Floating Point and IEEE-754 Compliance for NVIDIA GPUs part 2/3

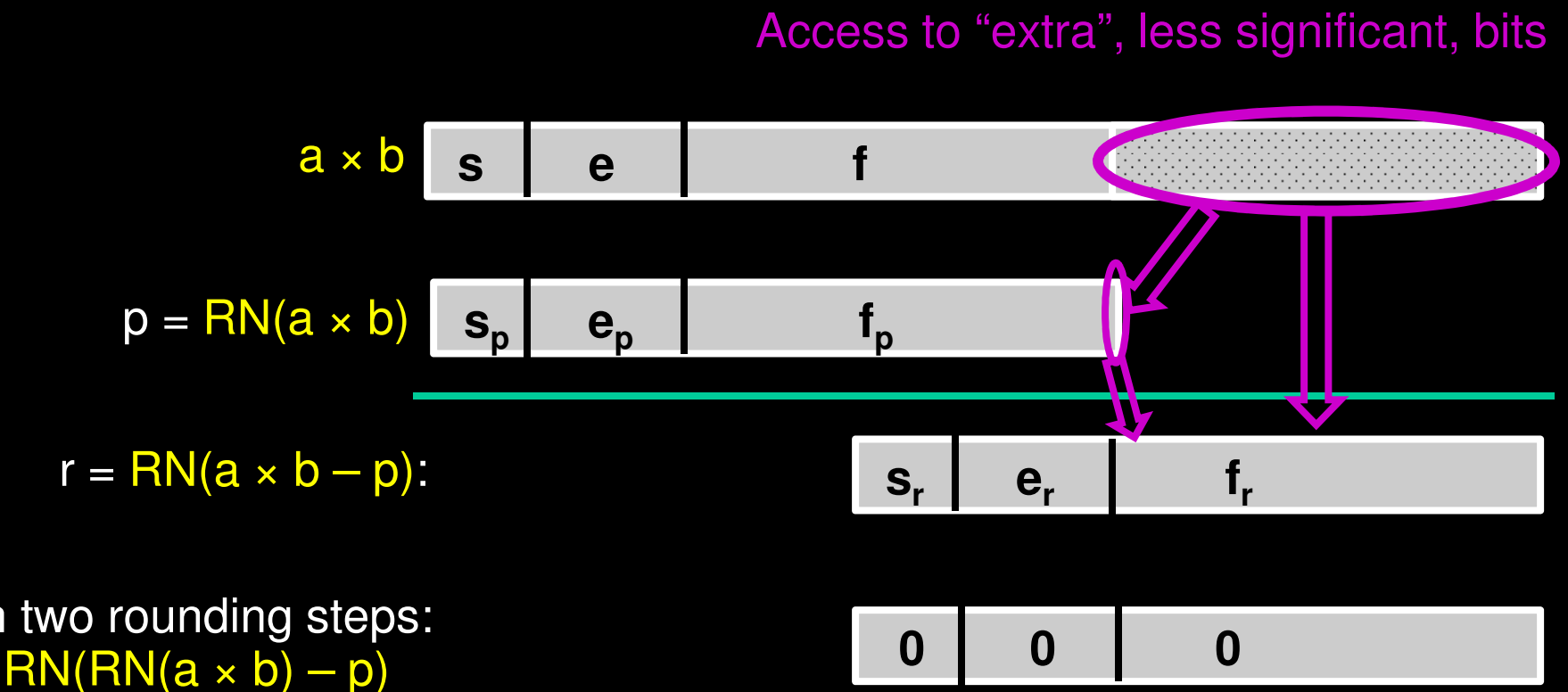
Nathan Whitehead

Alex Fit-Florea

NVIDIA

IEEE 754 Why FMA is a good thing

rnd(a × b + c): two operations, one rounding

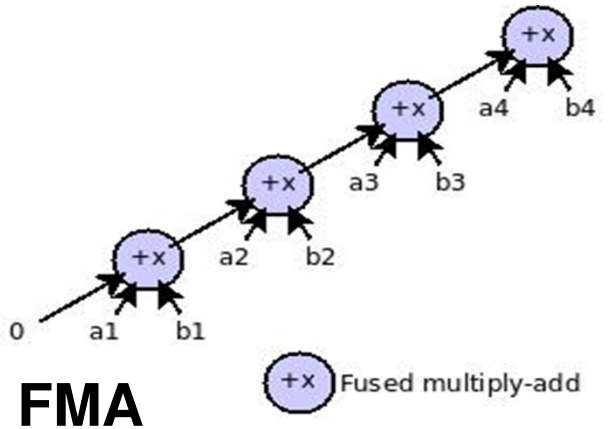


with two rounding steps:
q = RN(RN(a × b) – p)

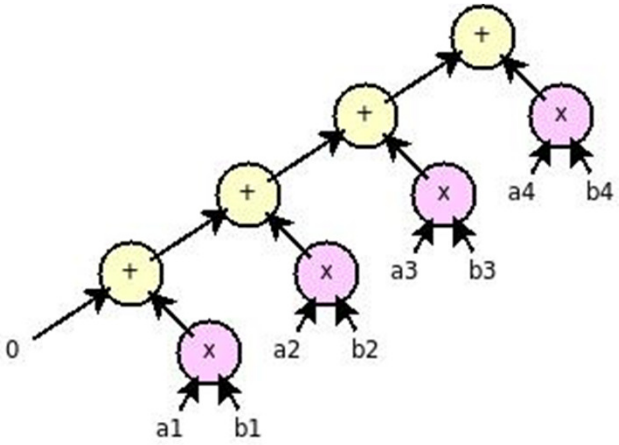
Computing Dot Product

Dot product – algorithms

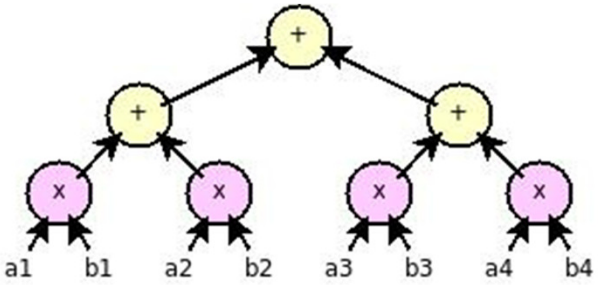
$$D = \sum_{i=1}^{i \le 4} a_i b_i$$



FMA



Serial



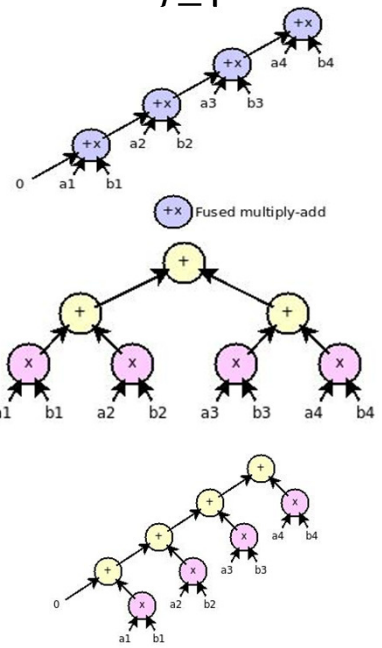
Parallel

Dot Product—Numerical Example

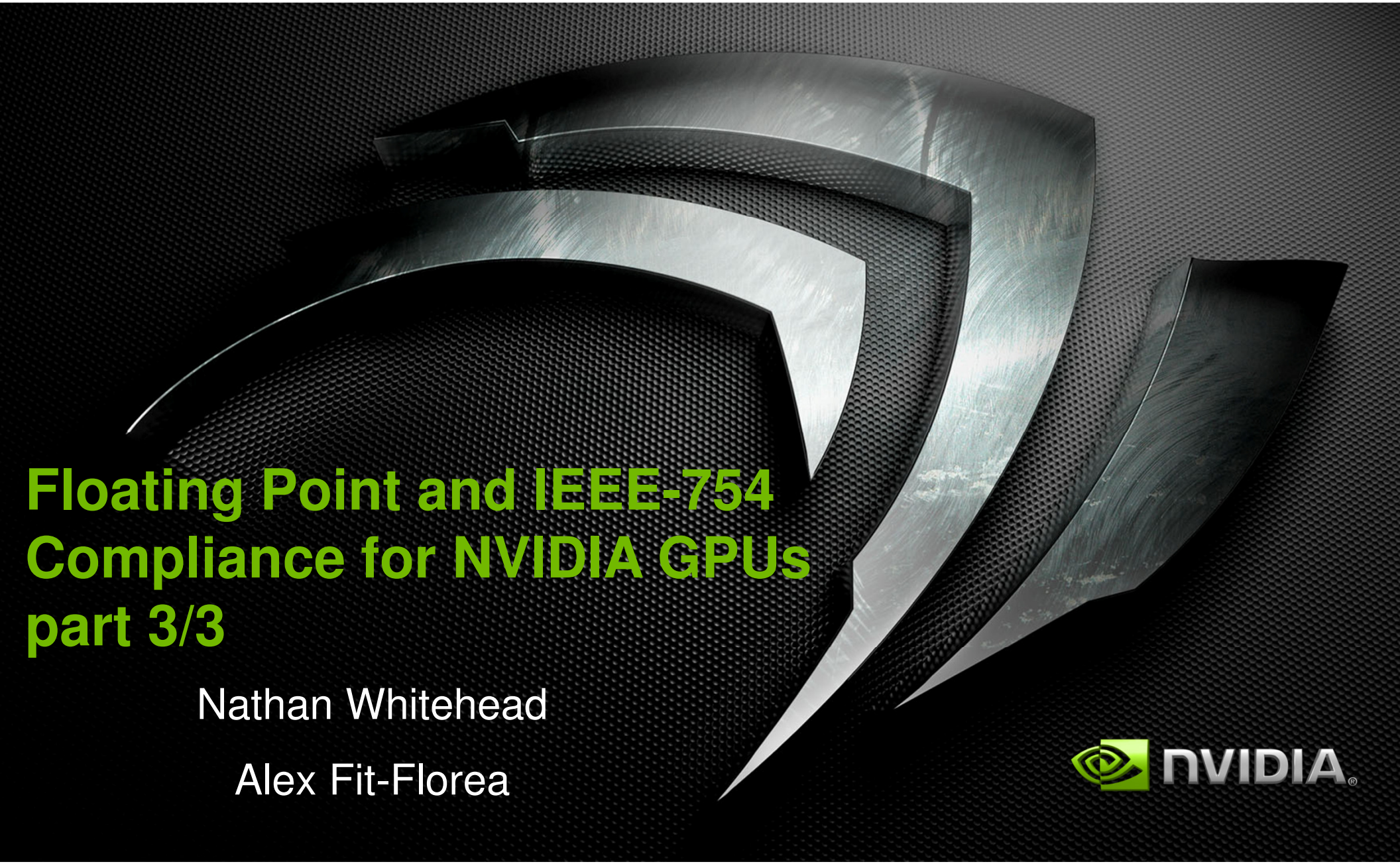
	a_i
1	1.907607
2	-.7862027
3	1.147311
4	.9604002

	b_i
1	-.9355000
2	-.6915108
3	1.724470
4	-.7097529

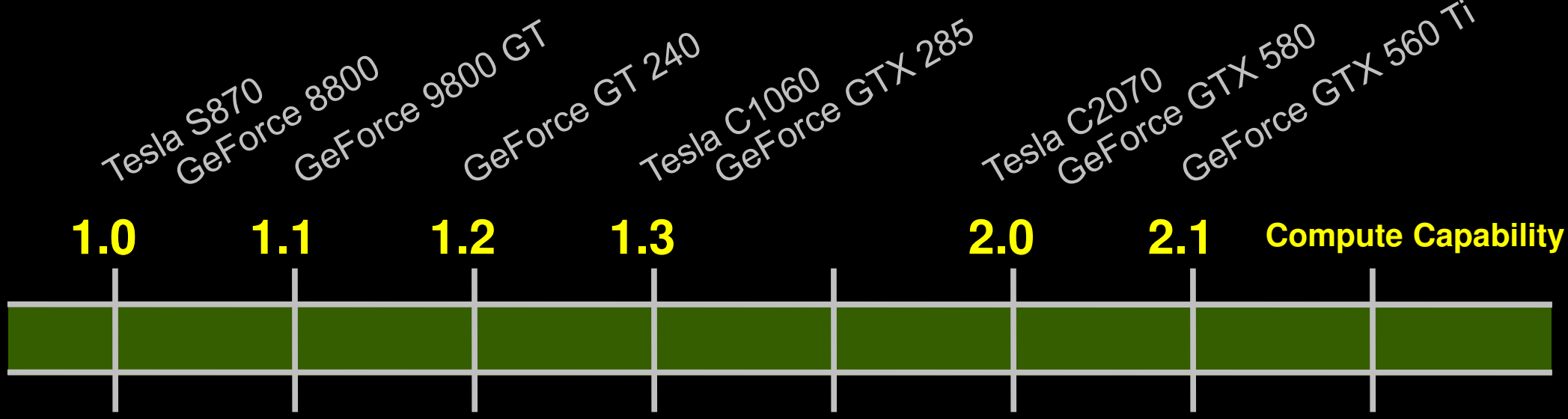
$$D = \sum_{i=1}^{i \le 4} a_i b_i$$



.0559587528…
.0559587515
.0559587478
.0559588074


Floating Point and IEEE-754 Compliance for NVIDIA GPUs part 3/3
Nathan Whitehead
Alex Fit-Florea
NVIDIA

Evolution of CUDA Numerics



Single precision only

Denormals FTZ
Approximate div sqrt

Single and double

FTZ
Approx.

IEEE 754
Correctly
Rounded
FMA

Single and double

IEEE 754
Correctly rounded div sqrt
FMA

CUDA Compiler Flags

Quick and Dirty compiling:

```
nvcc -arch=sm_10
            sm_13
            sm_20
```

Even better, learn `-gencode=...`

Default is 1.0, need specific targets to use new features (e.g. DP)

Compute capability 2.0 single precision

IEEE 754 mode (default)

```
--ftz=false
--prec-div=true
--prec-sqrt=true
```

Fast inaccurate mode

```
--ftz=true
--prec-div=false
--prec-sqrt=false
```

CUDA Compiler Intrinsics

`x + y` `__fadd_[rn|rz|ru|rd](x, y)`	addition
`x * y` `__fmul_[rn|rz|ru|rd](x, y)`	multiplication
`fmaf(x, y, z)` `__fmaf_[rn|rz|ru|rd](x, y, z)`	FMA
`1.0f / x` `__frcp_[rn|rz|ru|rd](x)`	reciprocal
`x / y` `__fdiv_[rn|rz|ru|rd](x, y)`	division
`sqrtf(x)` `__fsqrt_[rn|rz|ru|rd](x)`	square root

```
x = __fmul_rz(x, 0.99999f);
```

Why am I getting different numbers on the GPU and CPU???

Library function accuracy

A program to call `cos()`:

```
#include <stdio.h>
#include <math.h>

int main() {
    float x = 5992555.0f;
    printf("cos(%f): %.10g\n", x, cos(x));
}
```

```
gcc cos.c -lm -m64 && ./a.out
cos(5992555.000000): 3.3209046e-07
```

```
gcc cos.c -lm -m32 && ./a.out
cos(5992555.000000): 3.3209046e-07
```

Different libraries often give different results.

No guarantee of correct rounding

x87 and SSE

x87

Extended 80-bit precision in registers for computation

Control flag for *optionally* rounding to single/double precision

Memory storage format is typically IEEE 754 single and double precision

No FMA

Often default for 32-bit compilation

SSE

`float`

IEEE 754 single precision

`double`

IEEE 754 double precision

No FMA

Often default for 64-bit compilation

x87 and SSE (cont.)

sum.c using **`double`** declarations

```
gcc sum.c -m64 && ./a.out
s = 1.99999999917733362
```

This uses SSE, IEEE 754 double precision

```
gcc sum.c -m32 -O0 && ./a.out
s = 1.99999999917733362

gcc sum.c -m32 -O3 && ./a.out
s = 2.00000000000024425
```

Optimization level can affect precision

```
gcc sum.c -m32 -O3 -mpc64 && ./a.out
s = 1.99999999917733362
```

x87, rounded to *double*

```
gcc sum.c -m32 -O3 -mpc32 && ./a.out
s = 1.95367431640625000
```

x87, rounded to *single (?)*

Why am I getting different numbers on the GPU and CPU???

Math library functions not guaranteed to be identical

Order of operations matters, algorithm matters

FMA increases accuracy for GPU

Precision of CPU can be tricky with x87

Compute capability 1.0 – 1.3 have approximate operations in single precision

GPU and CPU won't be the same

Algorithms can compute the same mathematical quantity but be numerically different

SSE is much simpler