21st ECMWF workshop on High Performance Computing in Meteorology

On the Use of Different Arithmetic Precisions and Its Impact on Dynamic Systems

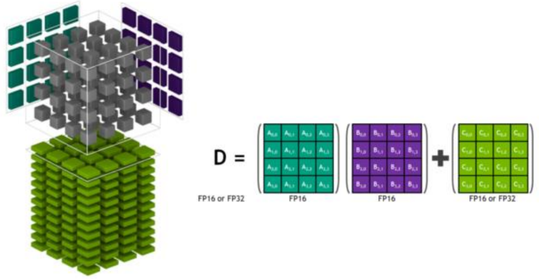
Florent DUGUET, Devtech HPC, NVIDIA fduguet@nvidia.com
ECMWF HPC Workshop, 16. September 2025

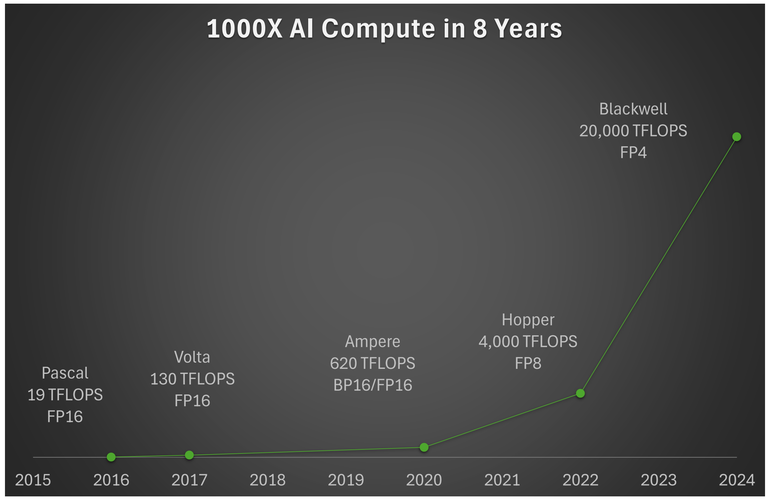
Motivation

From Graphics to AI

Building on 20+ years of Graphics Research

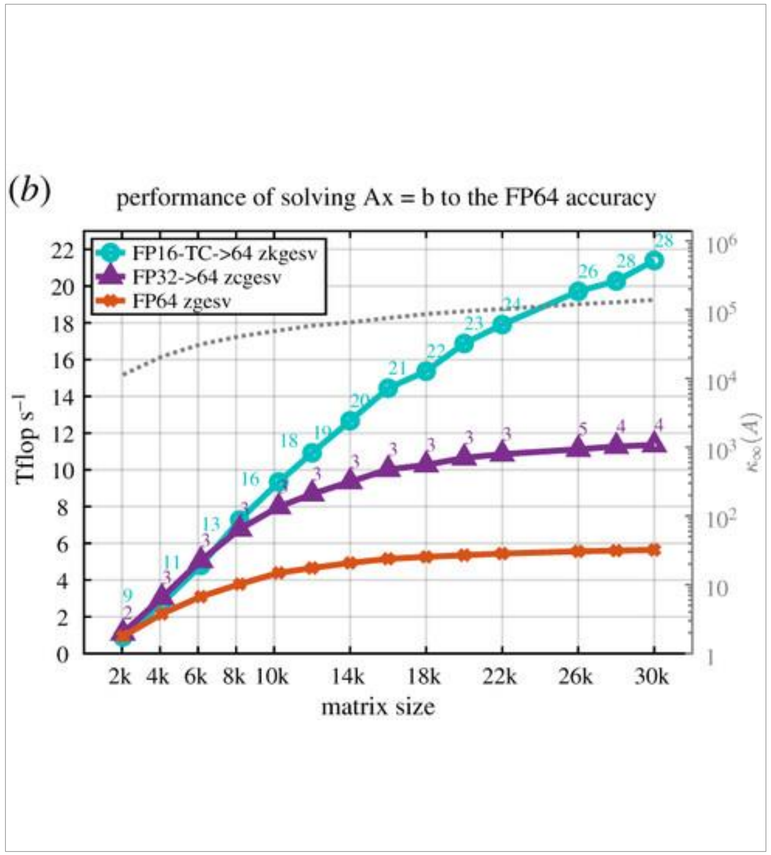
FP32 rules the Game





1000x AI Compute in 8 years

Significant performance gains have been obtained with **Tensor Cores** on reduced precision (FP16,FP8,FP4)



Opportunities for Mixed-Precision

Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. Haidar et al. 2020

Motivation

Floating-point in Weather Simulations

Spectral Transforms Matrix Multiply Operations	Many Weather Codes Double Precision still used	Ensemble Methods Stochastic Rounding

Leveraging Tensor Cores

- Tensor Core performance driven by innovation for AI
- Can we use lower precision higher throughput Tensor Cores to effectively compute higher precision ?

Using FP32 for scalar

- FP32 performance continues improving
- Can we use FP32 arithmetic to compute in higher precision ?
- Is FP64 hardware a requirement to achieve high precision computations ?

Storing in FP16 for Memory Footprint

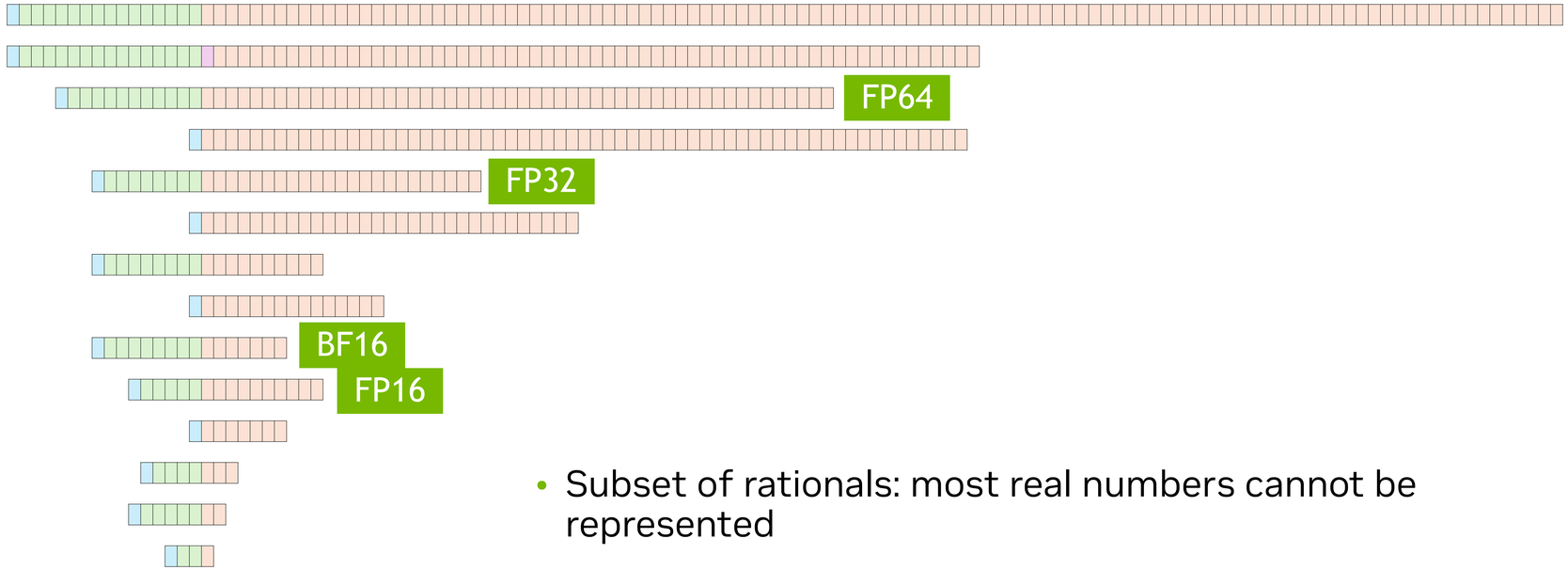
- Using FP16 as a storage format allows a reduced memory usage, and a higher effective memory bandwidth (more entries / s).
- Can we mitigate quantization issues with Stochastic Rounding ?

Matrix Operations – Leveraging Tensor Cores

Floating-Point is Quantized

x = 1.m * 2^e

			EPSILON
Emulation Library	IEEE-754 - binary 128	E15M112	1.93E-34
	FP80 (x8087)	E15M63	1.08E-19
FMA Hardware	IEEE-754 - binary 64	E11M52	2.22E-16
	INT64	E0M63	1.08E-19
FMA Hardware	IEEE-754 - binary 32	E8M23	1.19E-07
	INT32	E0M31	4.66E-10
Tensor Core	NVIDIA -TF32	E8M10	9.77E-04
	INT16	E0M15	3.05E-05
	BFLOAT 16	E8M7	7.81E-03
	IEEE-754 - binary 16	E5M10	9.77E-04
	INT8	E0M7	1.00E+00
	FP8	E4M3	1.25E-01
	FP8	E5M2	2.50E-01
	FP4	E2M1	5.00E-01
Tensor Core			

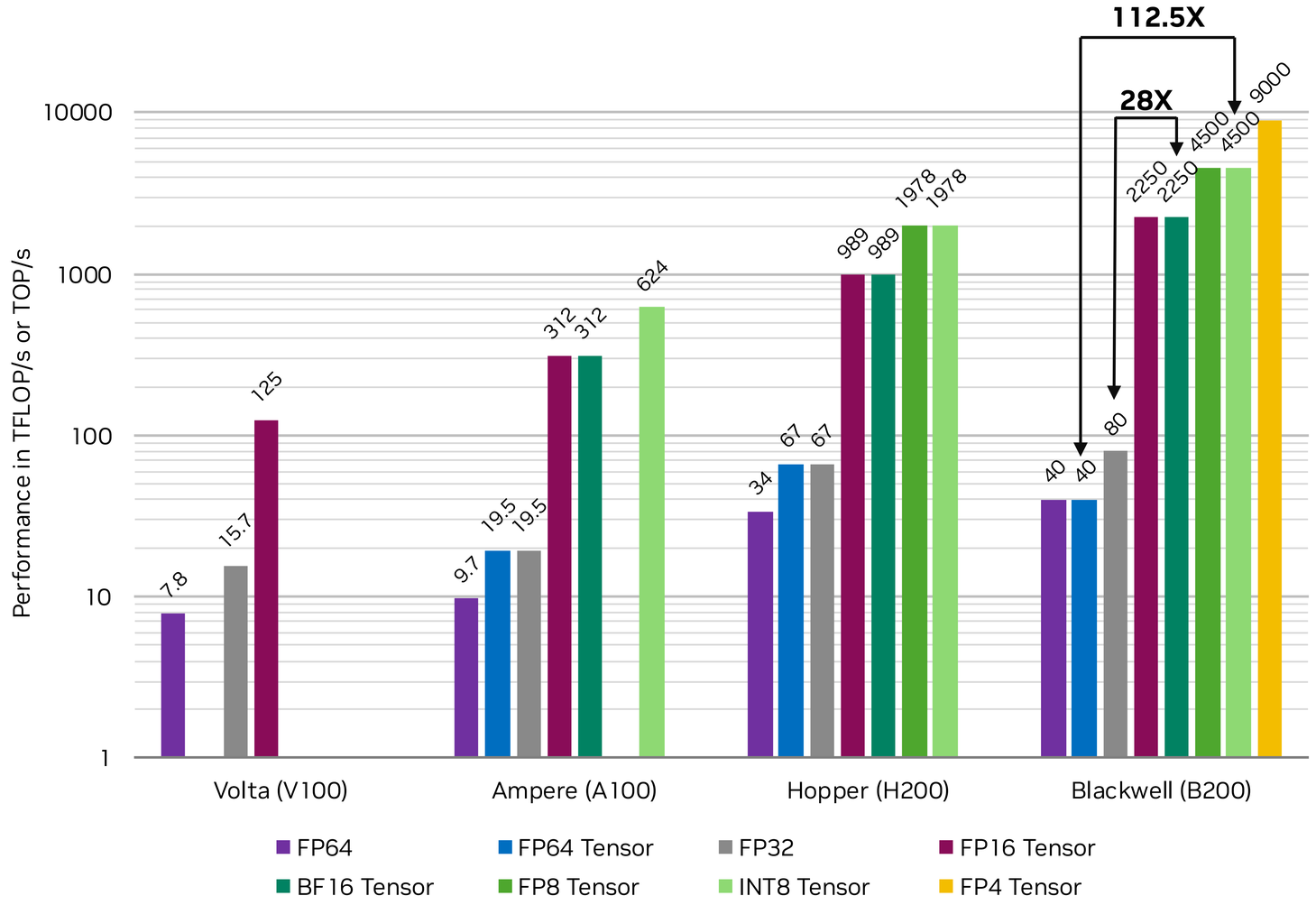


- Subset of rationals: most real numbers cannot be represented
- Non-commutative: operation re-ordering may lead to different result

Evolution of Peak Performance

Tensor Cores

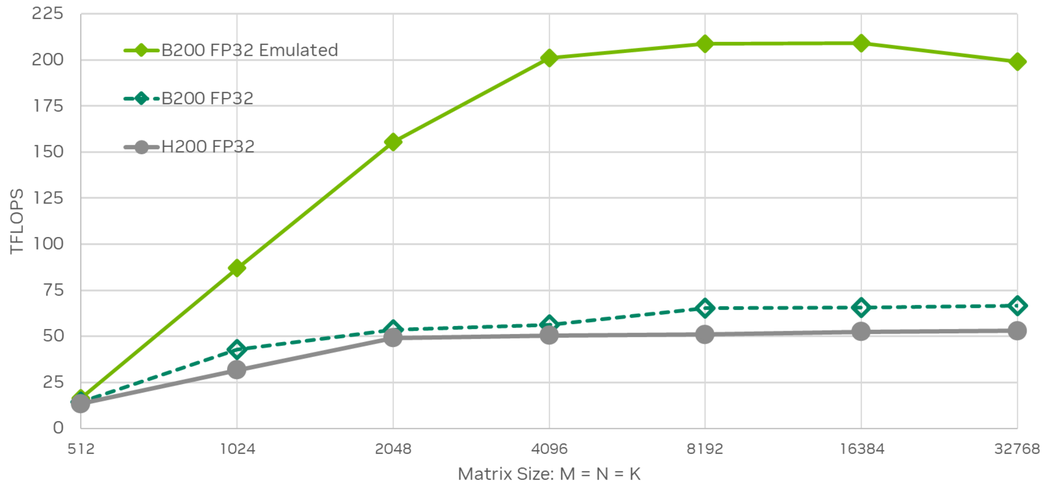
- Could we leverage Tensor Core performance to compute higher precision ?
- BF16 has 28x Throughput vs FP32
- INT8 has 110x Throughput vs FP64



More details on S72434 – GTC 2025: How Math Libraries Can Help Accelerate Your Applications on Blackwell GPUs

GEMM with Emulation

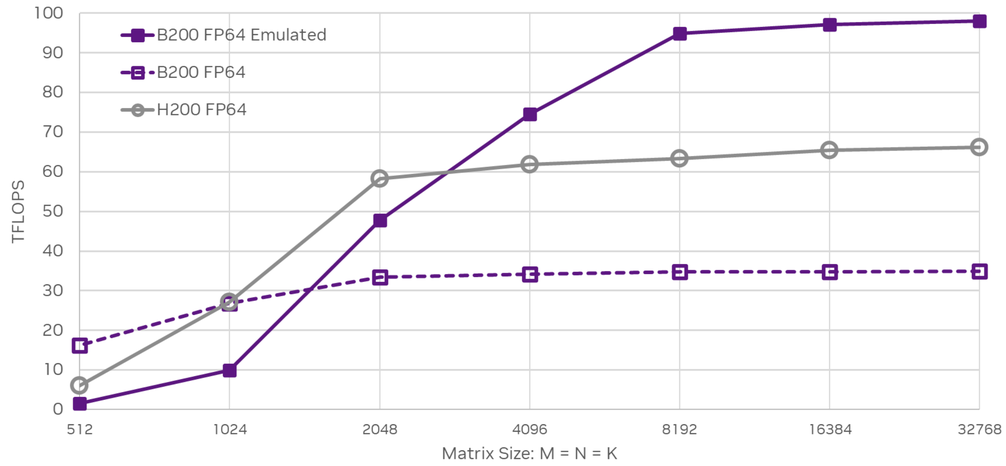
- FP32 using BF16 Tensor Cores [1]
- Available in cuBLAS :
  - CUBLAS_FP32_EMULATED_BF16X9_MATH, CUBLAS_COMPUTE_32F_EMULATED_16BFX9 – SGEMM computation with bfloat16 Tensor Cores



3x speed-up vs native FP32 on B200

4x speed-up vs FP32 on H200

- FP64 using INT8 Tensor Cores [2]
- Available soon in cuBLAS



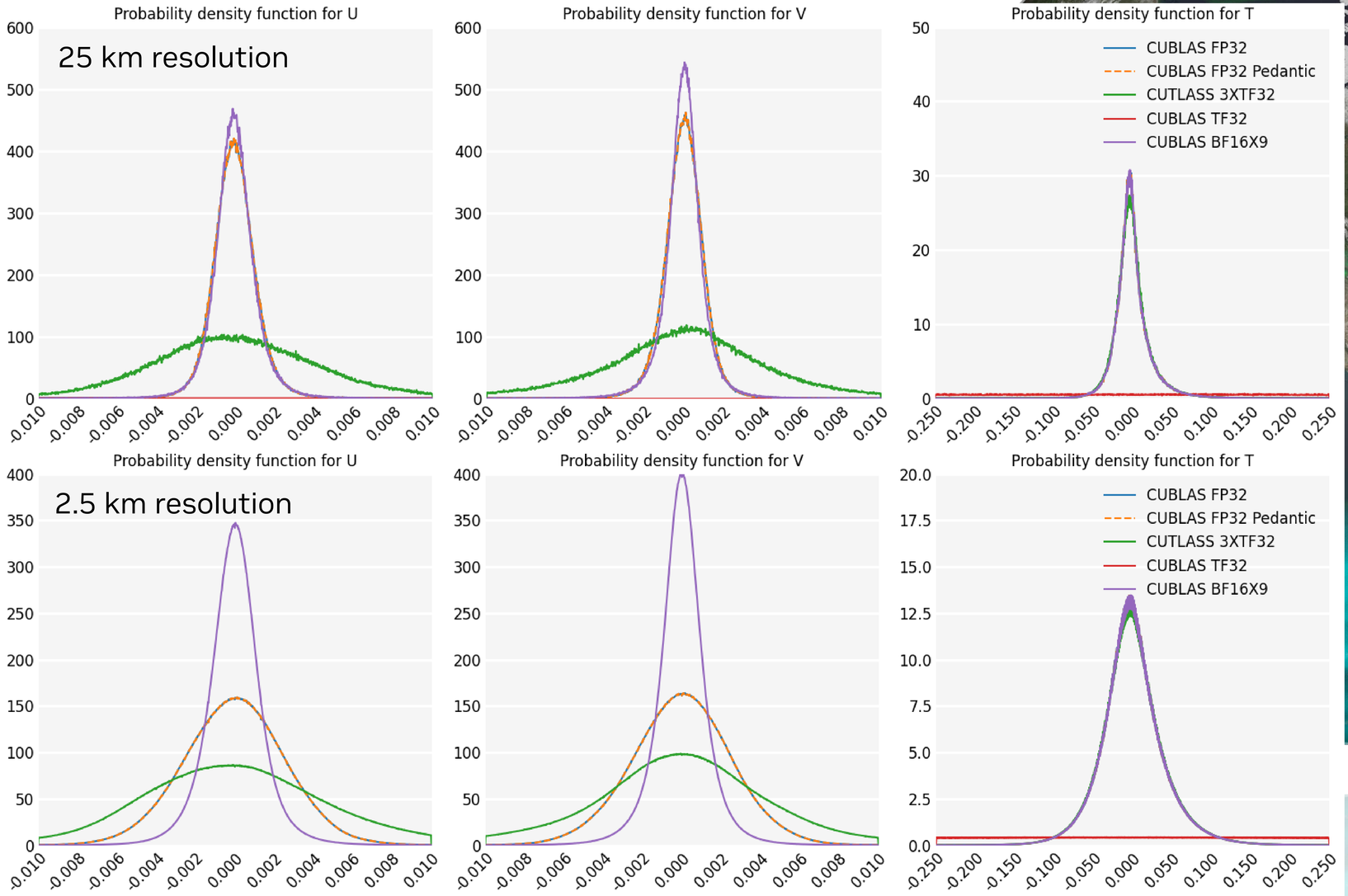
2.8x speed-up vs native FP64 on B200

1.5x speed-up vs FP64 on H200

1 https://arxiv.org/pdf/2203.03341 and https://arxiv.org/pdf/1904.06376
2 https://arxiv.org/abs/2306.11975 and https://arxiv.org/abs/2409.13313
See GTC Session Energy-Efficient Supercomputing Through Tensor Core-Accelerated Mixed-Precision Computing and Floating-Point Emulation [S71487]

Accelerating Weather Simulation On B200 with BF16x9

FP32 emulation brings 2.4x reduction in GEMM runtime



Global spherical harmonics transforms library underpinning the IFS

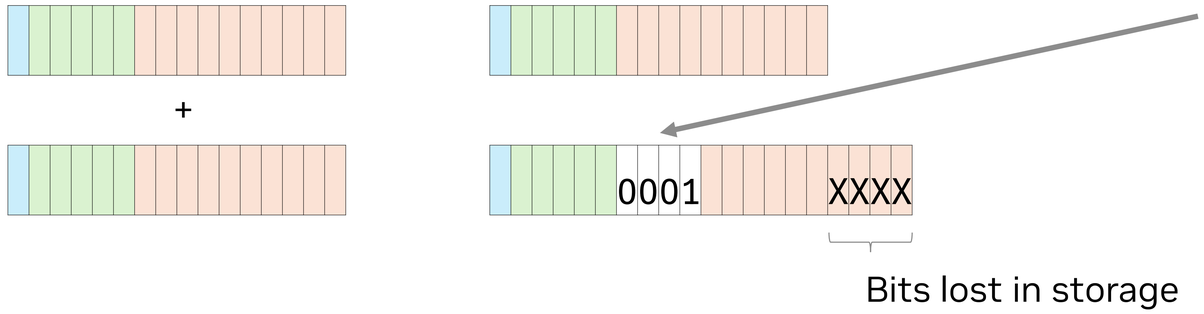
For more details on Spectral Transforms workflow, please see Lukas Mosimann presentation "Efficient Spectral Transforms On NVIDIA Hardware" of this Workshop.

Better Leveraging FP32

Floating Point Arithmetic

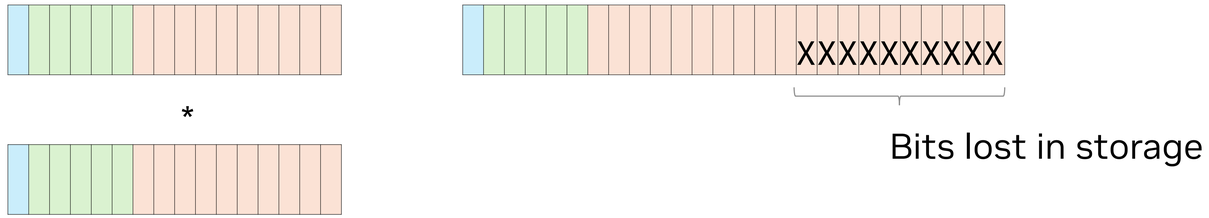
Add and Mul

- For the **addition**, exponents need to be “aligned” so mantissa to be added



Mantissa is shifted
(inserting leading 1 for normal)

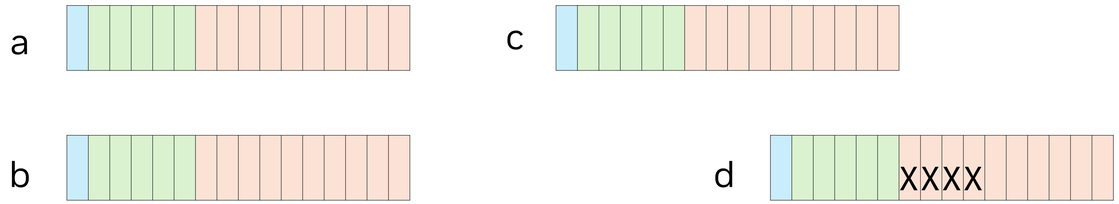
- For the **multiplication**, lower-order bits cannot be stored



Exact Arithmetic

Mul and Add can be represented exactly with two floats
TwoSum / TwoProd

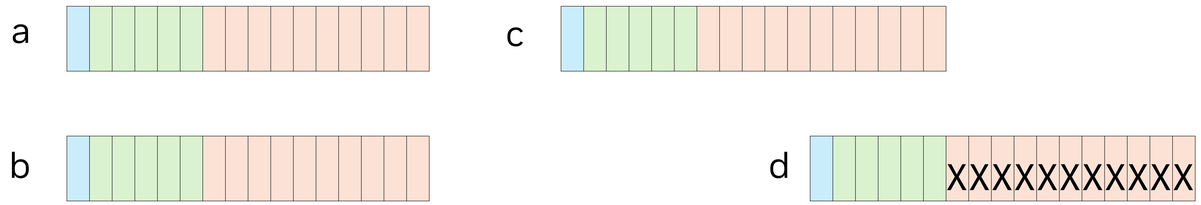
- For the **addition**, a+b = c+d, where d represents the “lost” bits



ALGORITHM 1: – Fast2Sum(a, b).

$$s \leftarrow \mathrm{RN}(a + b)$$
$$z \leftarrow \mathrm{RN}(s - a)$$
$$t \leftarrow \mathrm{RN}(b - z)$$

- For the **multiplication**, a*b = c+d, where d represents the “lost” bits



ALGORITHM 3: – Fast2Mult(a, b).

$$\pi \leftarrow \mathrm{RN}(a \cdot b)$$
$$\rho \leftarrow \mathrm{RN}(a \cdot b - \pi)$$

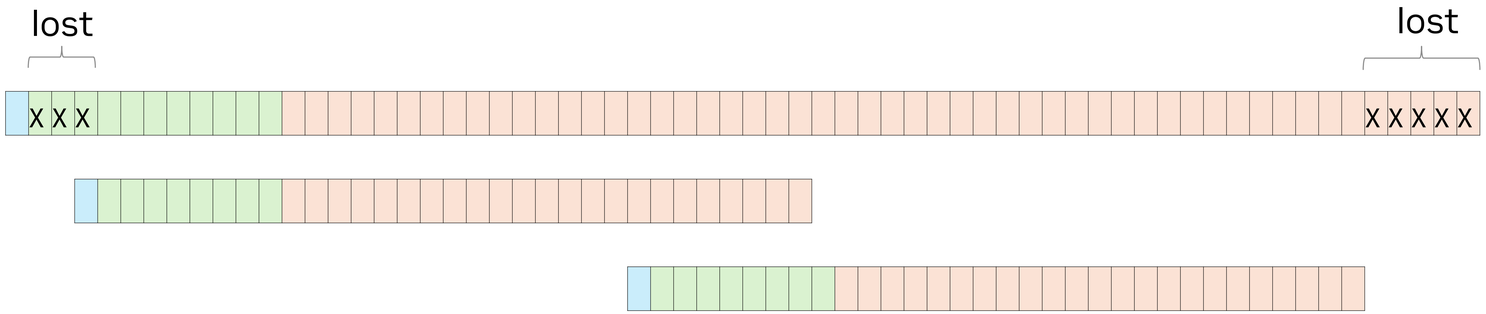
References:

[1] : Emulation of 3Sum, 4Sum, the FMA and the FD2 instructions in rounded-to-nearest floating-point arithmetic. [Graillat et al.] 2024
[2] : Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. [Joldes et al.] 2024

Double Word

Two FP32 to represent a single value

- Double Word arithmetic allows approximation of higher precision using two lower precision values



- Same Memory Footprint

ALGORITHM 6: – AccurateDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$.

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
2: $(t_h, t_\ell) \leftarrow \text{2Sum}(x_\ell, y_\ell)$
3: $c \leftarrow \text{RN}(s_\ell + t_h)$
4: $(v_h, v_\ell) \leftarrow \text{Fast2Sum}(s_h, c)$
5: $w \leftarrow \text{RN}(t_\ell + v_\ell)$
6: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(v_h, w)$
7: **return** (z_h, z_ℓ)

ALGORITHM 12: – DWTimesDW3$(x_h, x_\ell, y_h, y_\ell)$.

1: $(c_h, c_{\ell 1}) \leftarrow \text{2Prod}(x_h, y_h)$
2: $t_{\ell 0} \leftarrow \text{RN}(x_\ell \cdot y_\ell)$
3: $t_{\ell 1} \leftarrow \text{RN}(x_h \cdot y_\ell + t_{\ell 0})$
4: $c_{\ell 2} \leftarrow \text{RN}(t_{\ell 1} + x_\ell \cdot y_h)$
5: $c_{\ell 3} \leftarrow \text{RN}(c_{\ell 1} + c_{\ell 2})$
6: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(c_h, c_{\ell 3})$
7: **return** (z_h, z_ℓ)

References:
[1] : Emulation of 3Sum, 4Sum, the FMA and the FD2 instructions in rounded-to-nearest floating-point arithmetic. [Graillat et al.] 2024
[2] : Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. [Joldes et al.] 2024

Quantitative Finance Example

Mitigation of FP64 performance reduction with Algorithmic Change

- Some financial products are valued with the following formula:

$$P = e^{-r_1 T} - e^{-r_2 T}$$

- Using a naïve implementation may lead to errors, however, some algorithmic change allows better precision:

r_1	0.50%
r_2	0.80%
T	1/365

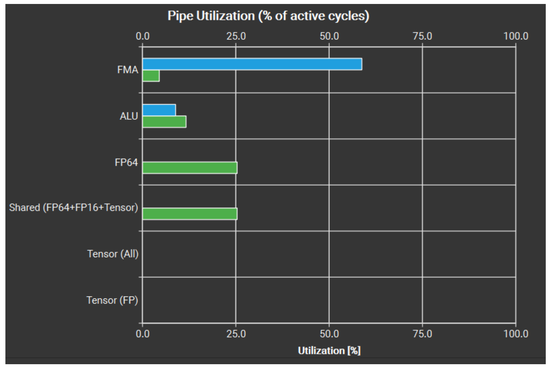
Implem.	-	P	err vs ref
FP64	naïve	-8.21932445238537E-06	-7.0E-12
FP32	naïve	-8.22544097900390E-06	7.4E-04
DW64	naïve	-8.21932445305151E-06	7.4E-11
FP32	expm1	-8.21932553662918E-06	1.3E-07
DW64	expm1	-8.21932445244295E-06	-1.2E-15
FP64	expm1	-8.21932445244296E-06	-

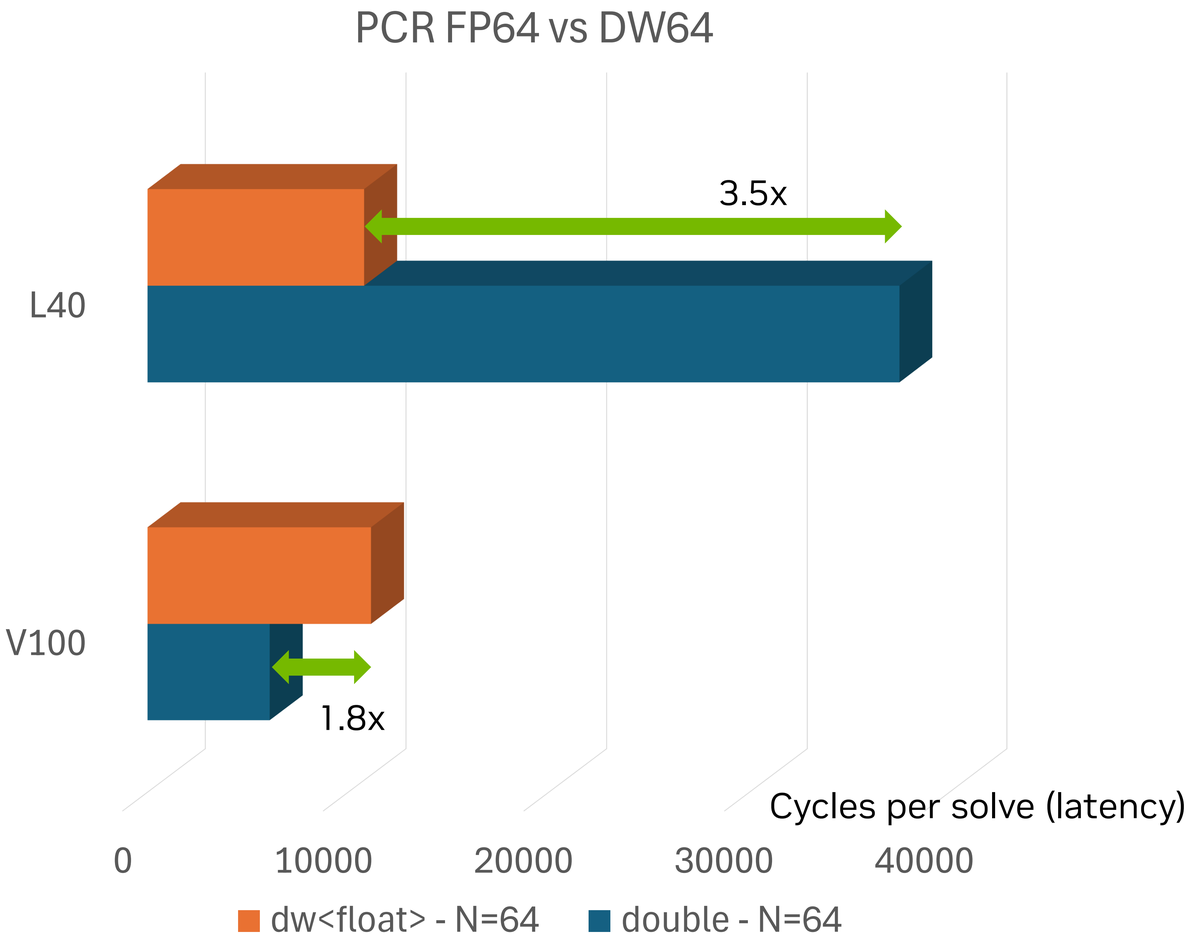
Combining algorithmic change (**`expm1`** instead of **`exp`**), and `doubleword<FP32>`, we obtain better precision than a naïve implementation with FP64.

Tridiagonal Solver Example

Using Parallel Cyclic Reduction to Solve Tridiagonal System

- Comparing PCR solve of a 64 entries tridiagonal system with non-trivial terms between:
  - FP64 : regular built-in double precision
  - DW64 : implementation of double-word with two floats
- Results match up to 9.1e-15 = 2^-46.6, in line with DW64 precision expectation
- DW64 is about 2x slower than FP64 on V100
- DW64 is similar on V100 and L40
- DW64 does not use FP64 hardware





Preliminary research work – not a benchmark

Using FP16 as a Storage Format to **Reduce Memory Footprint**

Lorenz Attractor

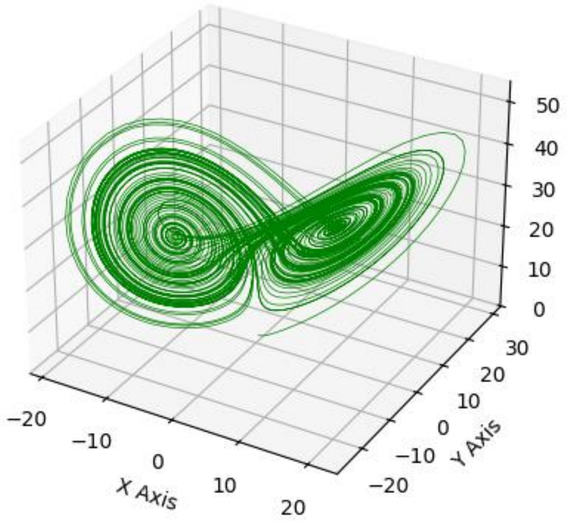
Python example (FP64)

$$\frac{dx}{dt} = \sigma(y - x) \qquad \sigma = 10$$

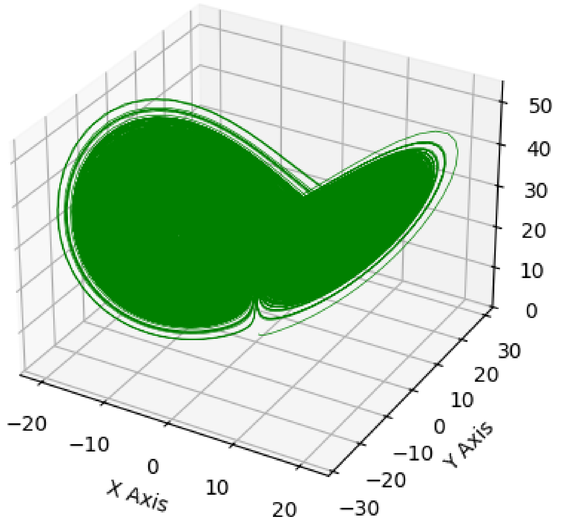
$$\frac{dy}{dt} = x(\rho - z) - y \qquad \rho = 28$$

$$\frac{dz}{dt} = xy - \beta z \qquad \beta = 8/3$$

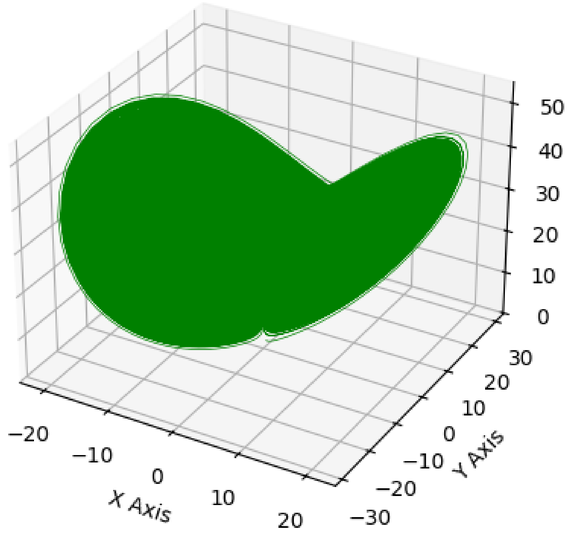
- Initial condition: [0,1,1.05]



10,000 iterations



100,000 iterations

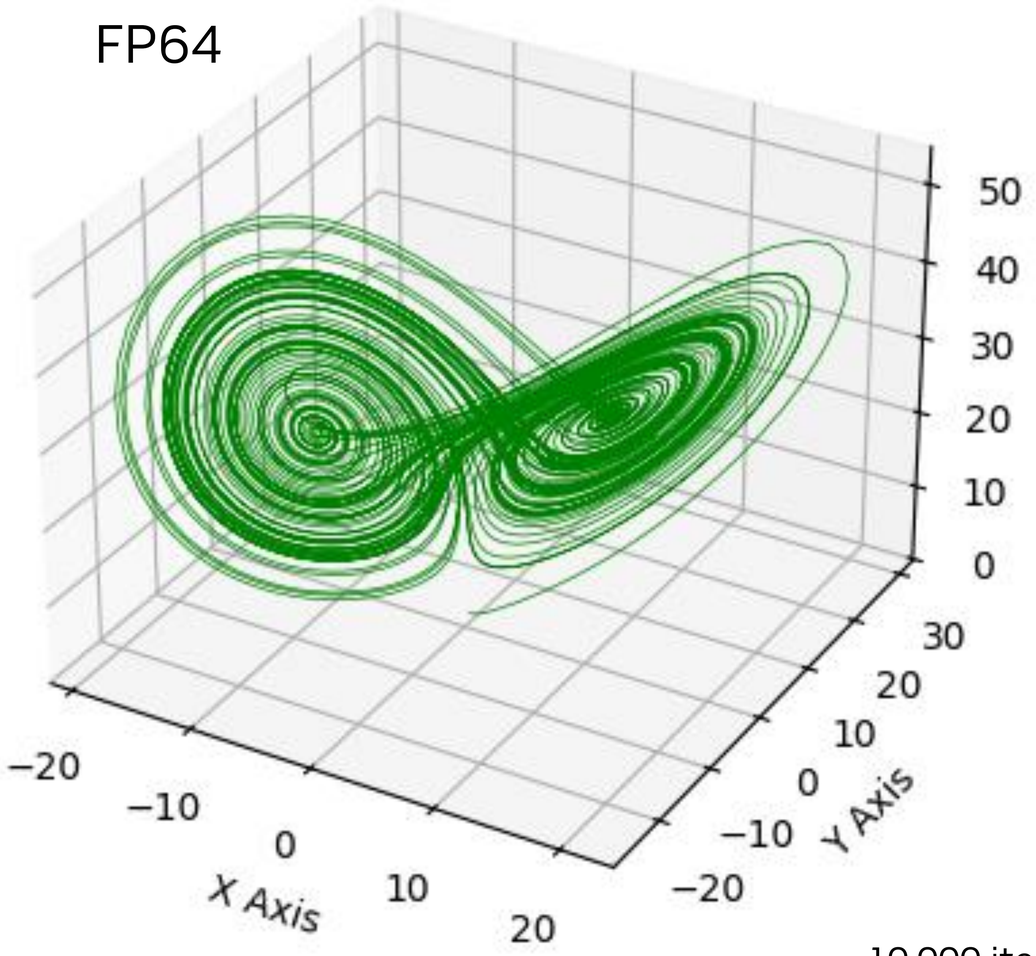


1,000,000 iterations

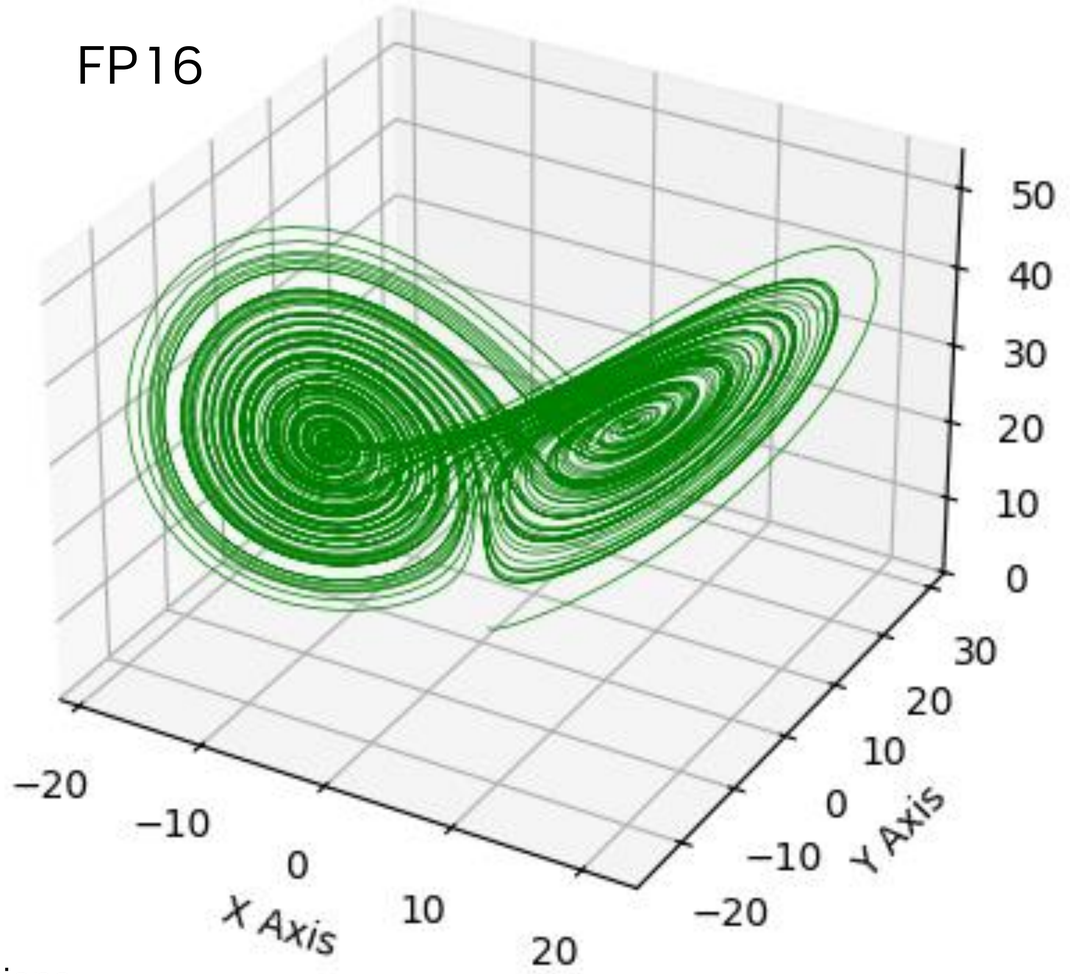
Lorenz Attractor

Running same in FP16 – [0,1,1.05]

FP64



FP16



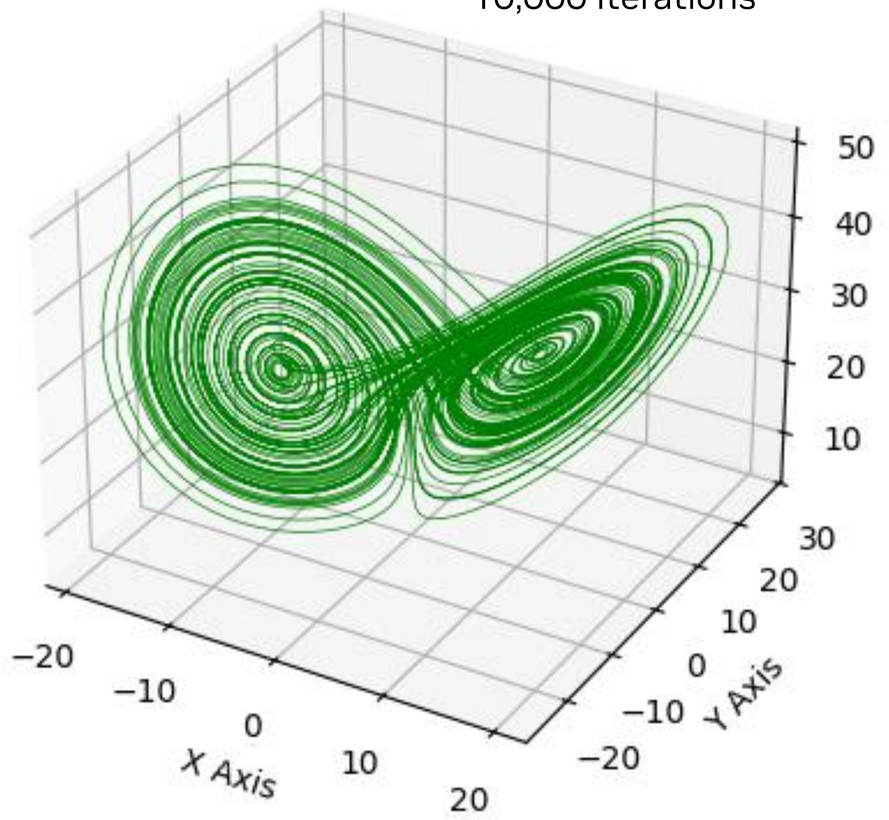
10,000 iterations

Lorenz Attractor

Running same in FP16 – init @ [0.48291016, 0.89599609, 14.21093750] = P-1381

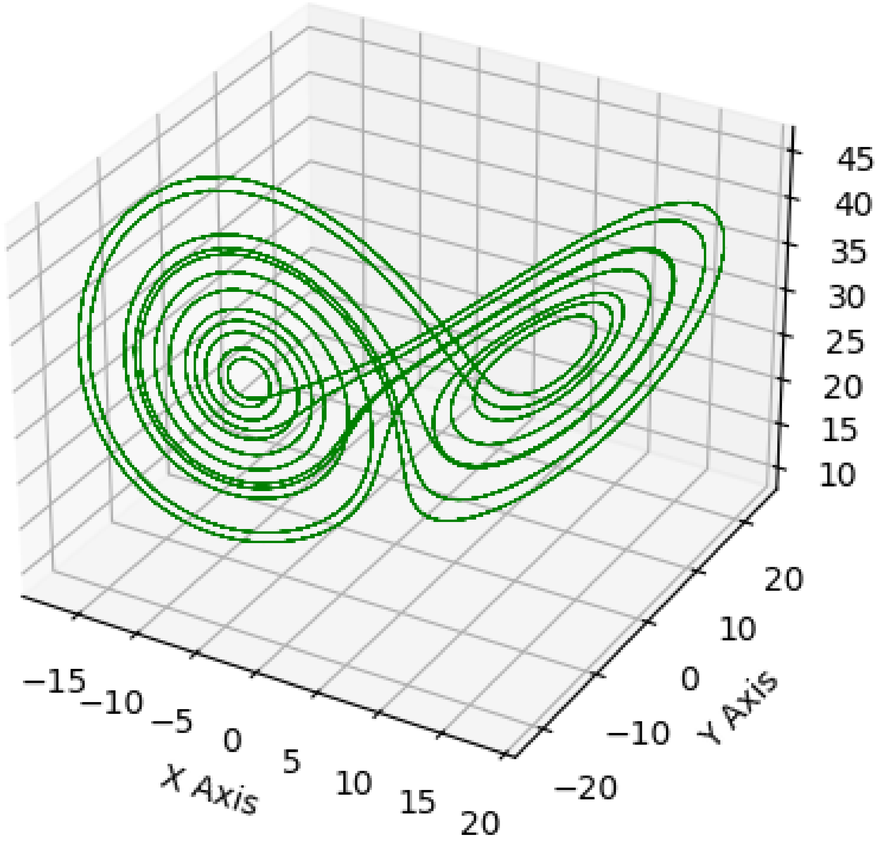
FP64

10,000 iterations



FP16

Looping @1381

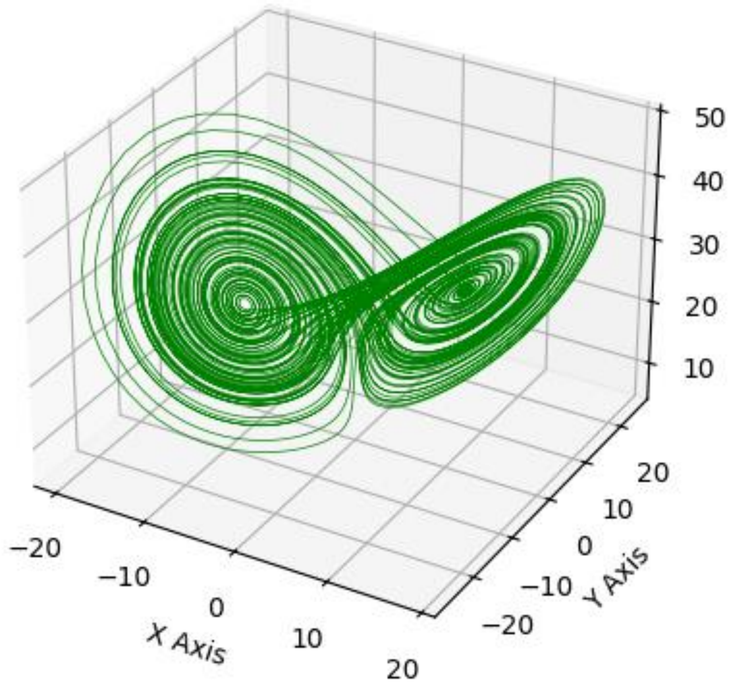


Lorenz Attractor

Running same in FP16 – init @ [8.45312500, 8.49218750, 26.92187500] = P-51

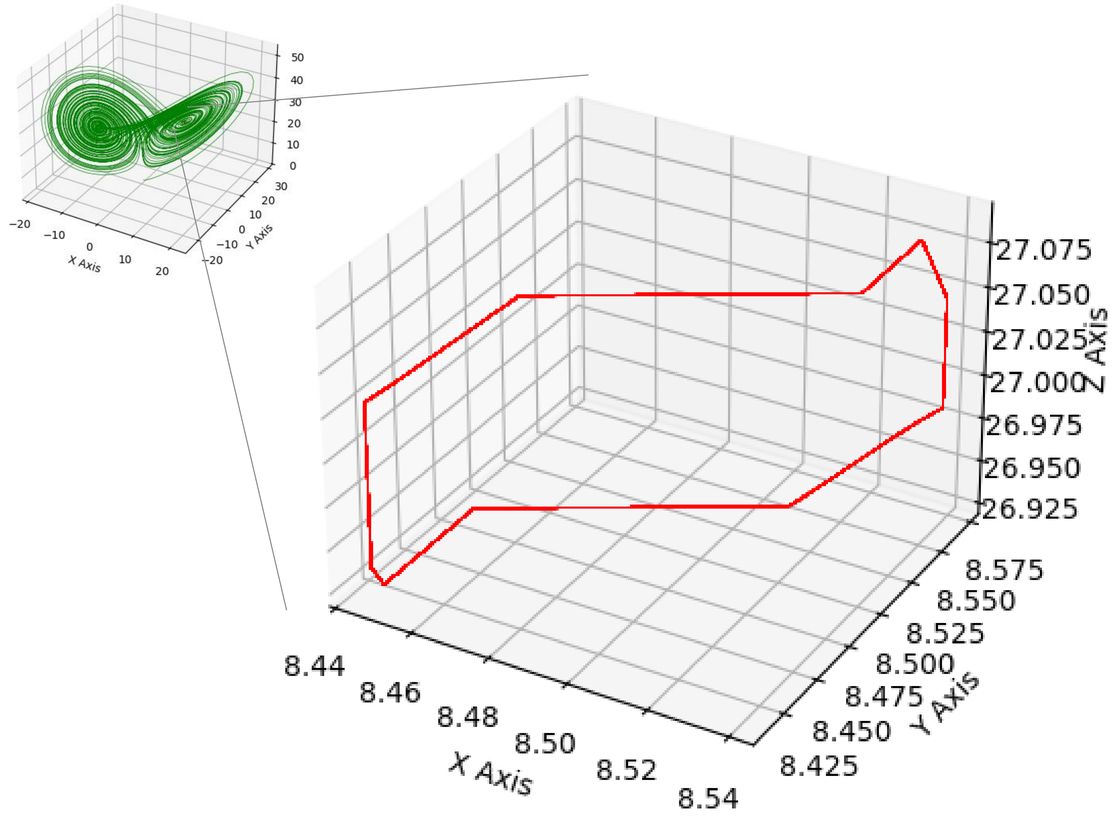
FP64

10,000 iterations



FP16

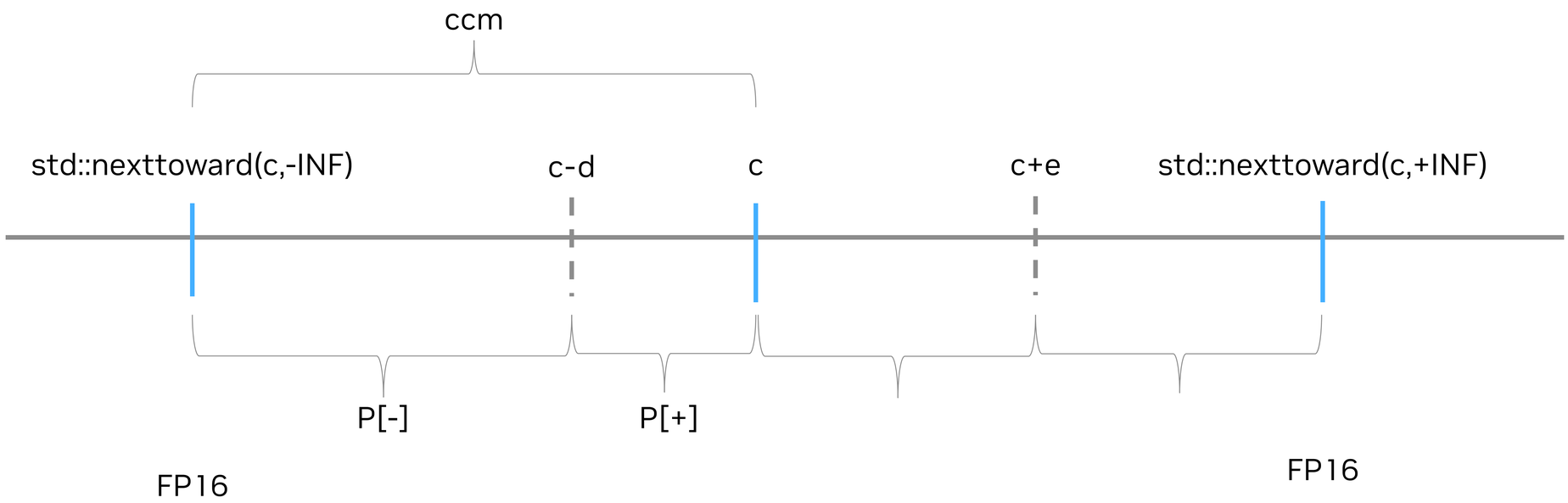
Looping @51



Stochastic Rounding

Stochastic Rounding

Rounding the results with a random number



- c-d and c+e cannot be represented by FP16
- c+ and c- (nexttoward) are closest representable FP16
- The probability of c+d to be rounded to c+ depends on d: P[+] = (1-d/ccm) – thus P[-] = d/ccm
- For each arithmetic operation, a random number needs to be sampled

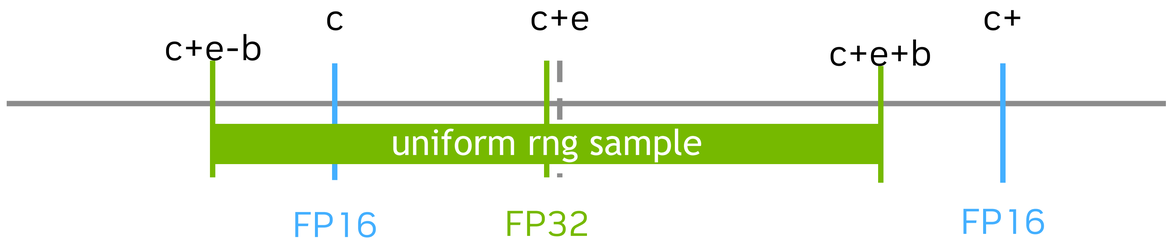
References

[1] : Climate Modeling in Low Precision: Effects of Both Deterministic and Stochastic Rounding. [Paxton et al.] 2022
[2] : Periodic orbits in chaotic systems simulated at low precision. [Klöwer et al.] 2023

Stochastic Rounding

CUDA implementation with FP32

- Algorithm:
  - We compute in FP32 e.g. c+e
  - We sample a uniform random number in]-b;b[where 2b is the FP16 epsilon, that we add to the result
  - Then, we round the result FP32 to FP16

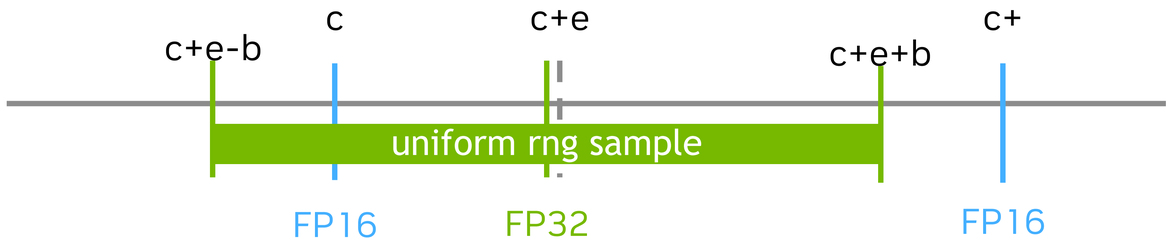



```
// fp_int_32 is union of float and int32
fp_int_32 funiform ;
// 1 + [0:8191]*2^-23
funiform.u = (uniform & 0x1FFF) | 0x3F800000 ;
funiform.f -= 1.0f ;

// hl is value to be rounded
fp_int_32 x ; x.f = hl ;

// extract b
fp_int_32 powerof2 ; powerof2.i = x.i & 0xFF800000 ;

// scale uniform to ]0;2b[
fp_int_32 adder ; adder.f = powerof2.f * funiform.f ;

// return _rz(c+e+x) which is _rn(c+e+x-b)
return __float2half_rz (x.f + adder.f) ;
```

Stochastic Rounding

CUDA implementation with FP32

- Algorithm:
  - We compute in FP32 e.g. c+e
  - We sample a uniform random number in]-b;b[where 2b is the FP16 epsilon, that we add to the result
  - Then, we round the result FP32 to FP16
- NOTES:
  - Each CUDA thread has a “state variable” that is the state of the generator – e.g. its **id**.
  - RNG type had little impact on the results [tested mrg32k3a and mcg]

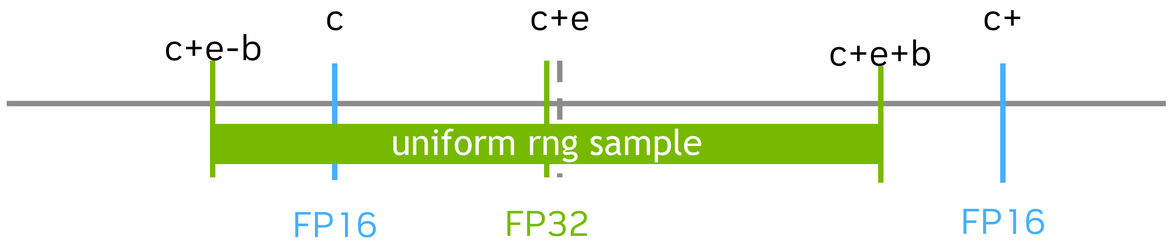



```
// fp_int_32 is union of float and int32
fp_int_32 funiform ;
// 1 + [0:8191]*2^-23
funiform.u = (uniform & 0x1FFF) | 0x3F800000 ;
funiform.f -= 1.0f ;

// hl is value to be rounded
fp_int_32 x ; x.f = hl ;

// extract b
fp_int_32 powerof2 ; powerof2.i = x.i & 0xFF800000 ;

// scale uniform to ]0;2b[
fp_int_32 adder ; adder.f = powerof2.f * funiform.f ;

// return _rz(c+e+x) which is _rn(c+e+x-b)
return __float2half_rz (x.f + adder.f) ;
```

Stochastic Rounding

CUDA implementation with FP32

- Algorithm:
  - We compute in FP32 e.g. c+e
  - We sample a uniform random number in]-b;b[where 2b is the FP16 epsilon, that we add to the result
  - Then, we round the result FP32 to FP16



- NOTES:
  - Each CUDA thread has a “state variable” that is the state of the generator – e.g. its **id**.
  - RNG type had little impact on the results [tested mrg32k3a and mcg]
  - Translates into 6 INT/FP32 instructions

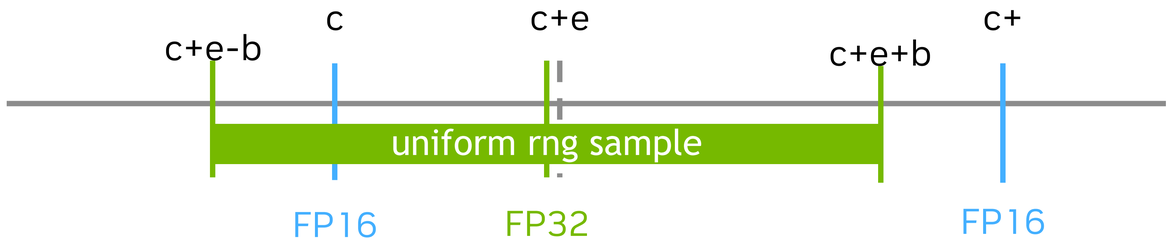
```
// fp_int_32 is union of float and int32
fp_int_32 funiform ;
// 1 + [0:8191]*2^-23
funiform.u = (uniform & 0x1FFF) | 0x3F800000 ;
funiform.f = 1.0f ;

// hl
fp_int

// ext
fp_int

// sca
fp_int_32 adder ; adder.f = powerof2.f * funiform.f ;

// return _rz(c+e+x) which is _rn(c+e+x-b)
return __float2half_rz (x.f + adder.f) ;
```

```
LOP3.LUT R5, R5, 0x1fff, RZ, 0xc0, !PT
LOP3.LUT R0, R4, 0xff800000, RZ, 0xc0, !PT
LOP3.LUT R3, R5, 0x3f800000, RZ, 0xfc, !PT
FADD R3, R3, -1
FFMA R0, R3, R0, R4
F2F.F16.F32.RZ R4, R0
```

Stochastic Rounding

On CC 10.0 - B100/B200

- Algorithm:
  - We compute in FP32 e.g. c+e
  - We sample a uniform random number in]-b;b[where 2b is the FP16 epsilon, that we add to the result
  - Then, we round the result FP32 to FP16



- NOTES:
  - Each CUDA thread has a “state variable” that is the state of the generator – e.g. its **id**.
  - RNG type had little impact on the results [tested mrg32k3a and mcg]
  - Translates into 6 INT/FP32 instructions
  - sm_100a (10.0) has a specific instruction

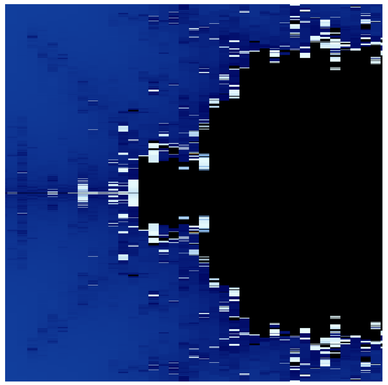
```
// compiled with -arch=sm_100a
#if __CUDA_ARCH_SPECIFIC__ == 1000
asm ("cvt.rs.f16x2.f32 %0, %1, %2, %3 ; " :
             "=r"(res.u32) :
             "f"(a),
             "f"(b),
             "r"(uniform)) ;
#endif
```

```
F2FP.F16.F32.PACK_AB.RS R6, R6, R7, R8
```

Stochastic Rounding

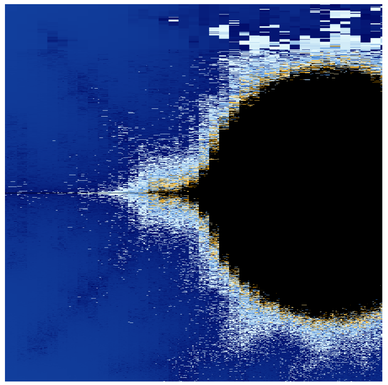
Mandelbrot Set – Feigenbaum points (4 and 9)

0.036



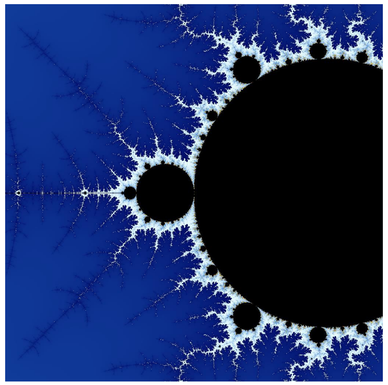
FP16

CenterIm	0
CenterRe	-1.3940462
Height	1024
Maxiter	4096
Precision	IEEE_FP64
Step	3.5623181221794055E-05
Width	1024

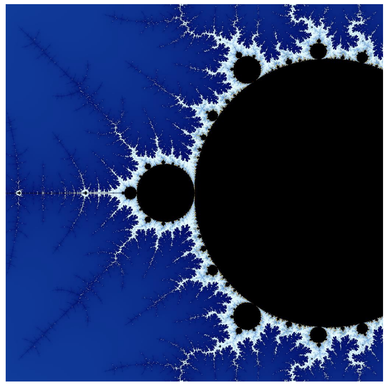


FP16 + SR

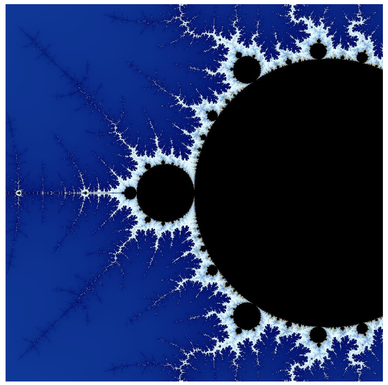
CenterIm	0
CenterRe	-1.401151982
Height	1024
Maxiter	131072
Precision	IEEE_FP64
Step	1.6051645790116297E-08
Width	1024



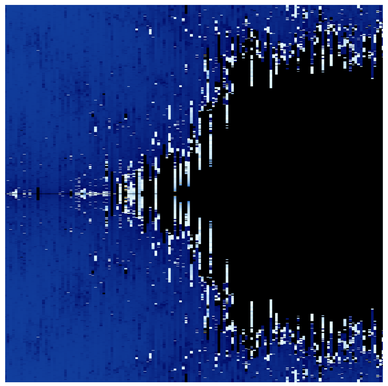
FP32

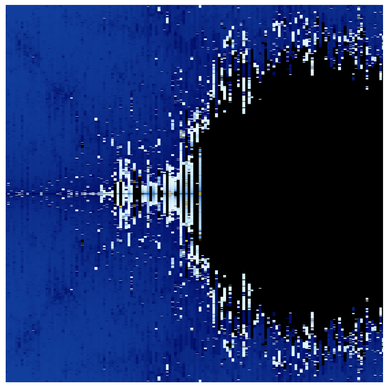


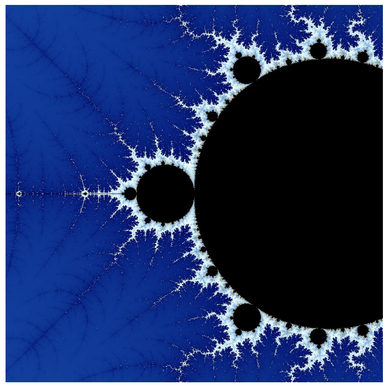
FP32 + SR



FP64







1.6 x 10^-5

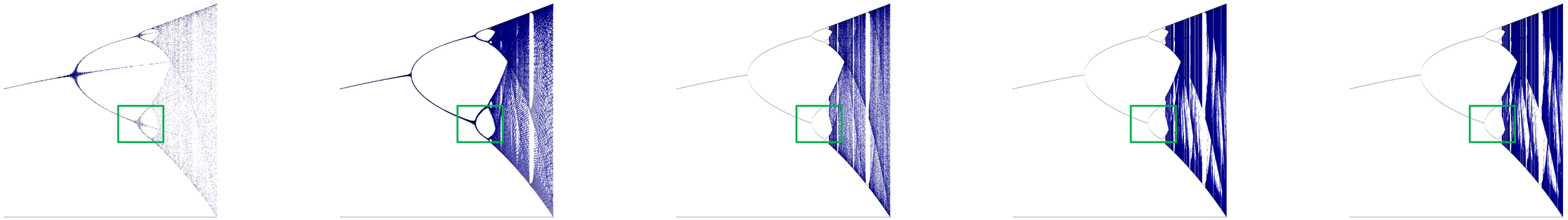
Using Stochastic Rounding for this Fractal does not help much for F9

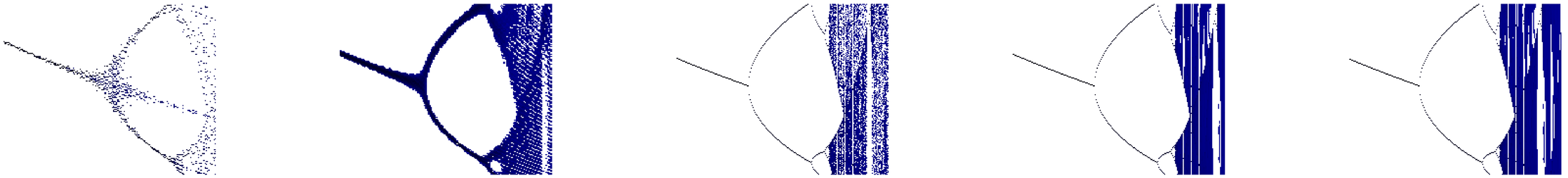
Stochastic Rounding

Illustration on the Logistic Map – see [Klöwer et al.] 2023

- Logistic map - $x_{n+1} = r x_n (1 - x_n)$

FP16 | FP16 + SR | FP32 | FP32 + SR | FP64





Erroneous attractor removed

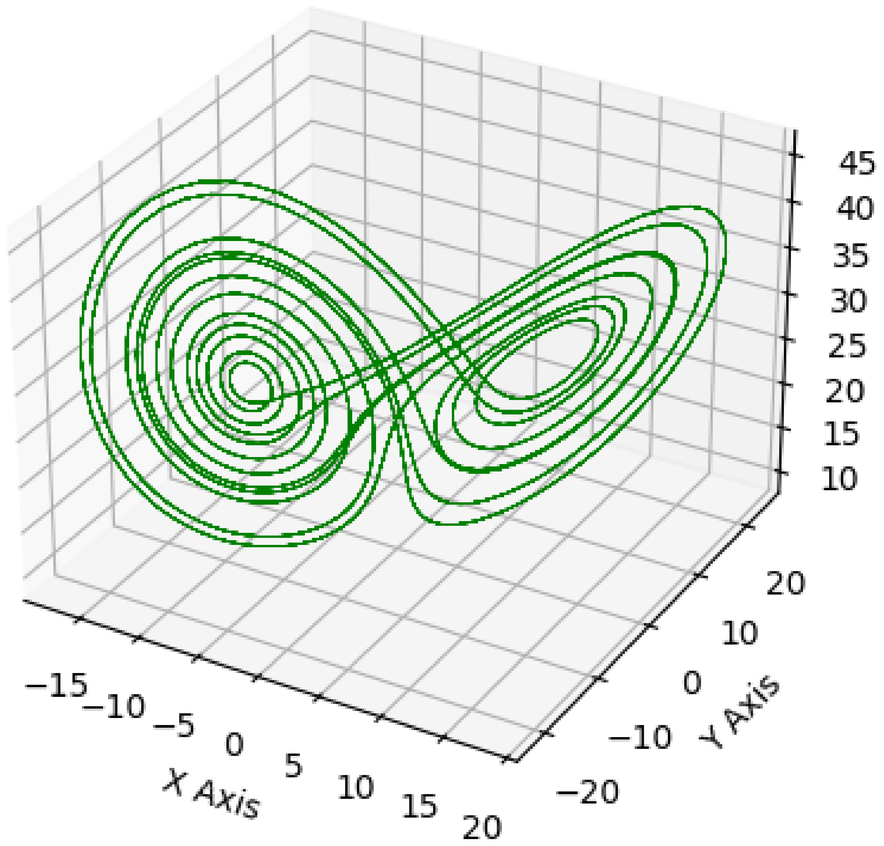
Denser sampling of chaotic areas

Stochastic Rounding

Lorenz Attractor with stochastic rounding @ P-1381

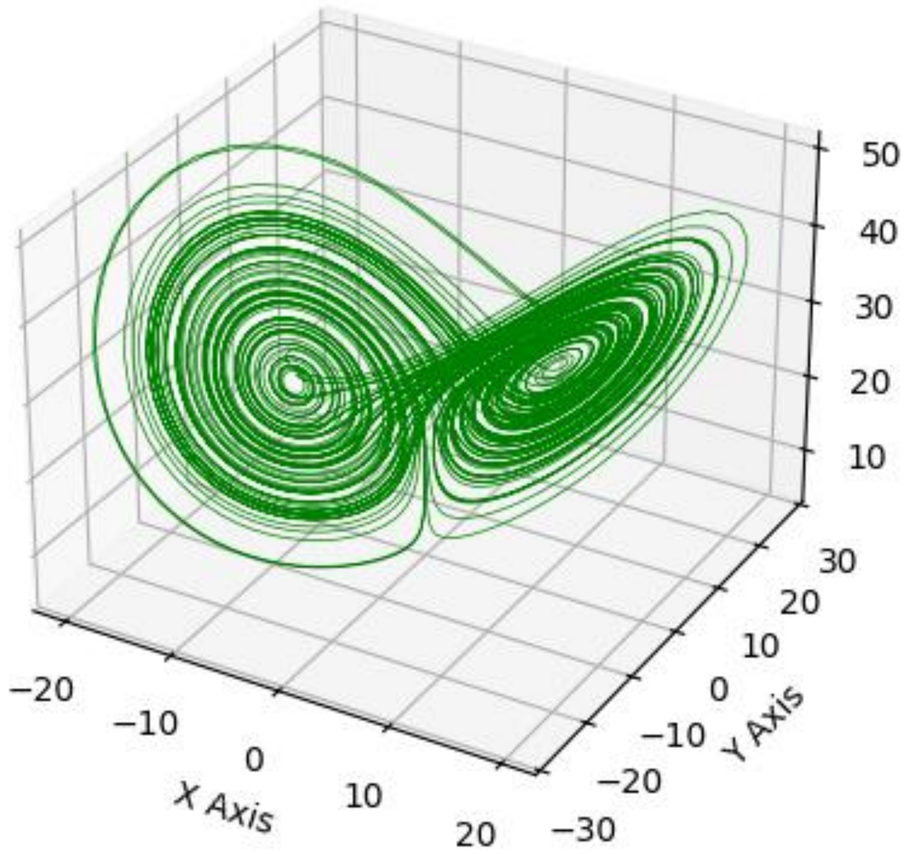
FP16

Looping @1381



FP16 + SR

10,000 iterations

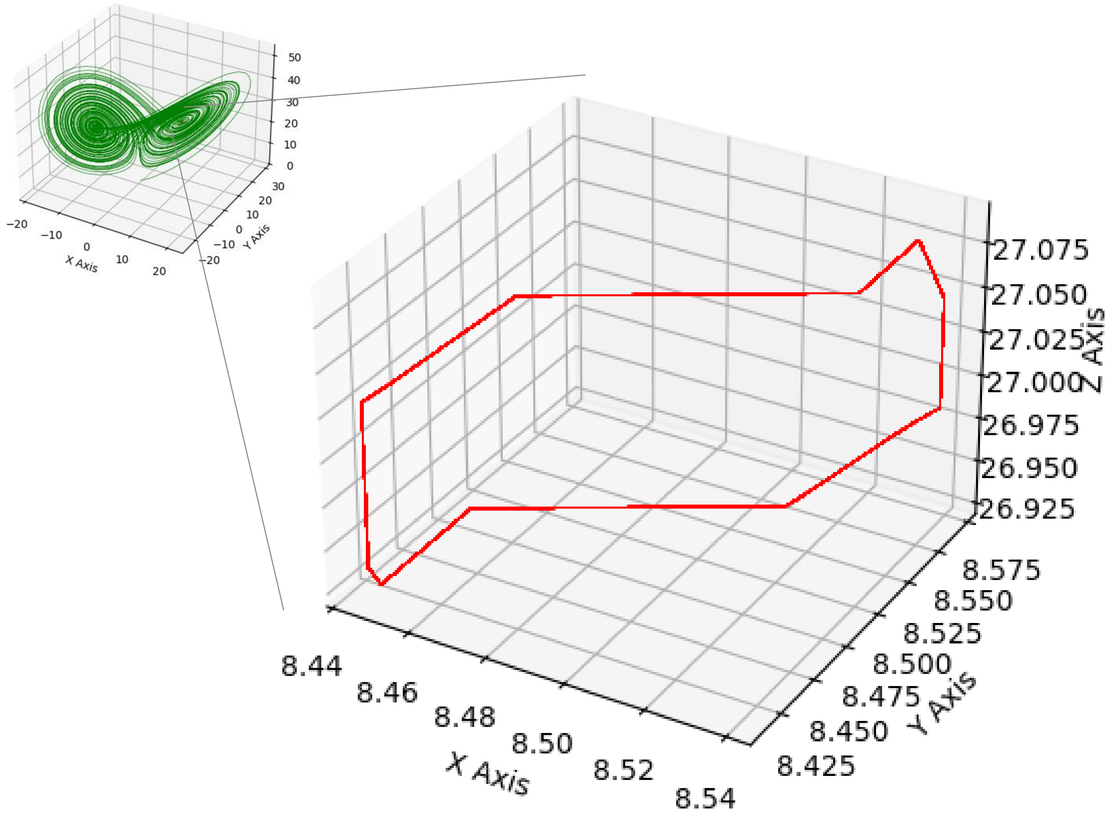


Stochastic Rounding

Lorenz Attractor with stochastic rounding @ P-51

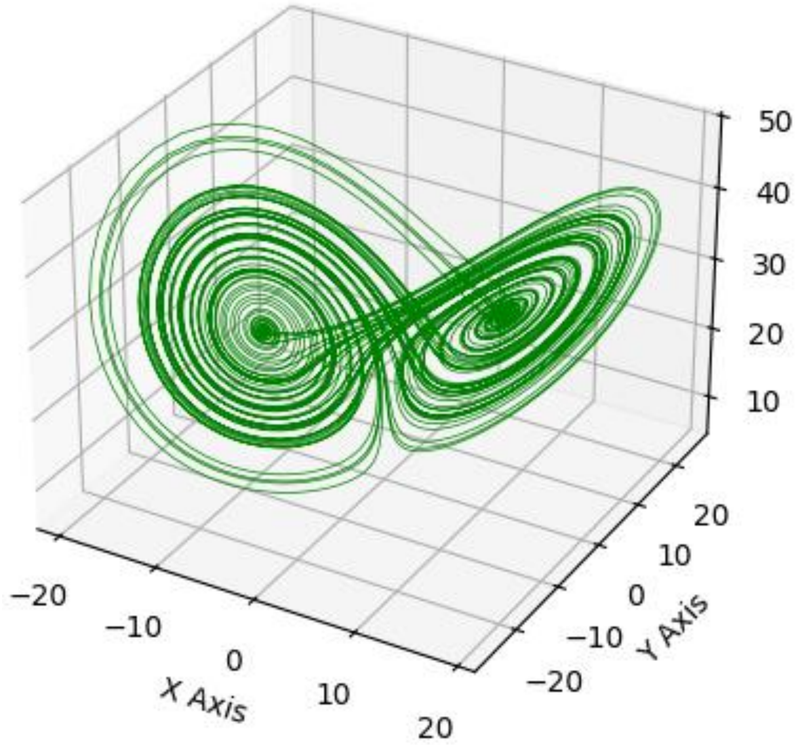
FP16

Looping @51



FP16 + SR

10,000 iterations

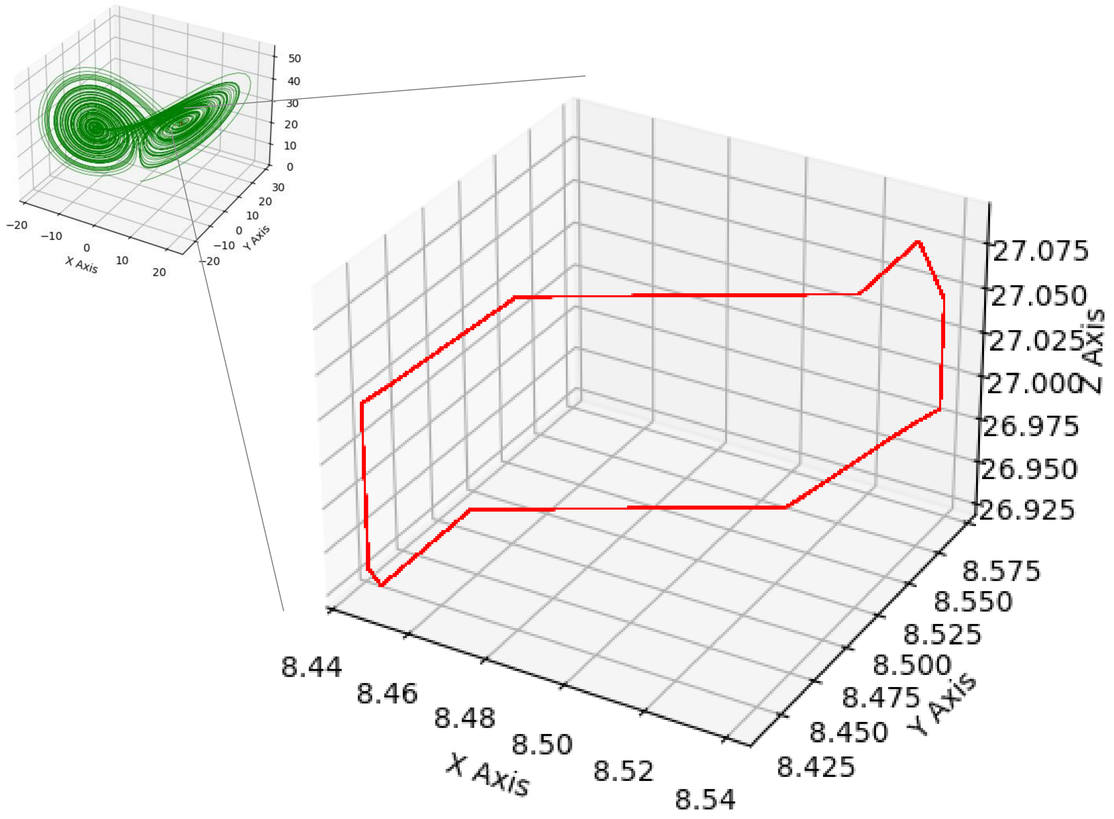


Stochastic Rounding

Lorenz Attractor with stochastic rounding @ P-51

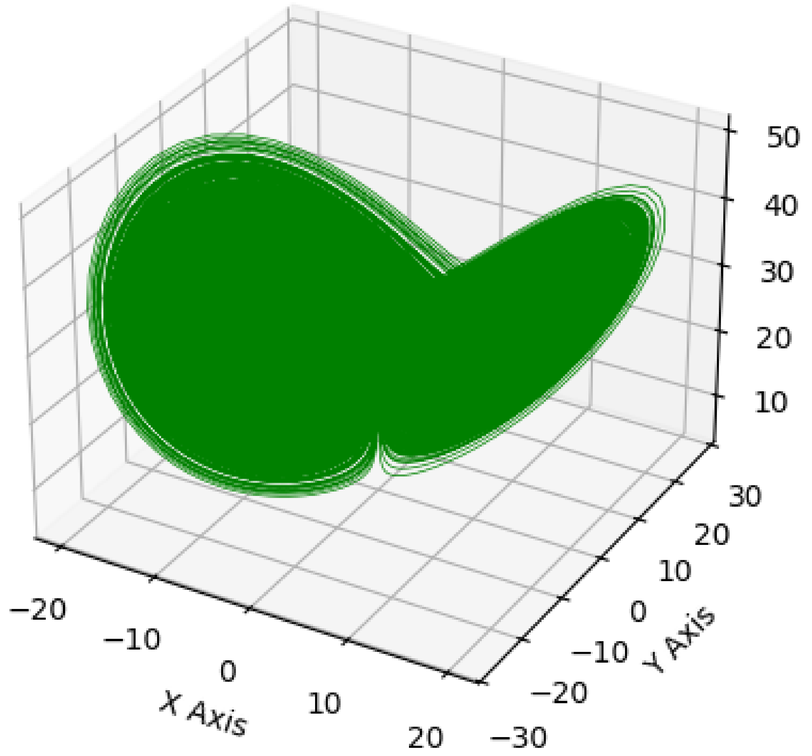
FP16

Looping @51



FP16 + SR

100,000 iterations



Going beyond single/double: Arithmetic on mixed precision hardware

- **Market trends determine hardware evolution**
  - Moving beyond single/double model
- **Success of FP32 in weather prediction shows potential impact**
  - Required more in-depth (numerical) analysis
- **FP64 accuracy sometimes needed**
  - But do we need it at high throughput in-silico?
- **Lots of work to be done**
  - Leverage FP16 and lower, stochastic rounding, mixed precision, improved utilization of emulation
- **Interested? Please reach out, we're happy to discuss/collaborate!**
  - E.g. tooling for testing lower precision?


NVIDIA
Questions ?