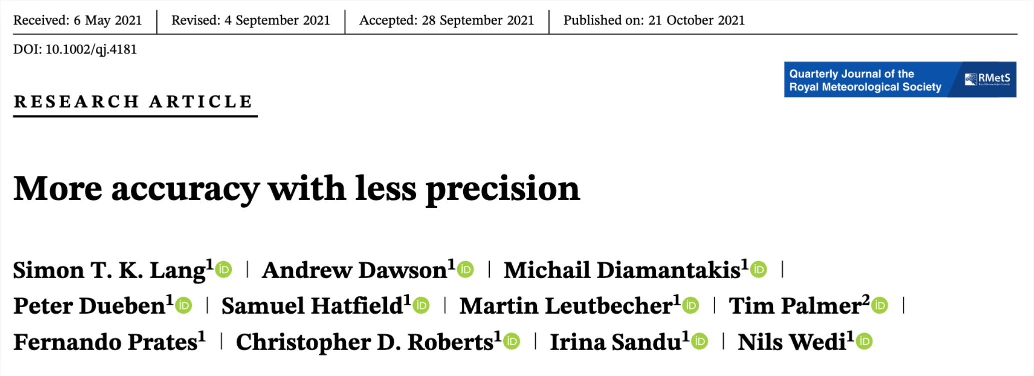
Reduced-precision computing for Earth-system modelling

Sam Hatfield

samuel.hatfield@ecmwf.int

ECMWF

“More accuracy with less precision”



Received: 6 May 2021 | Revised: 4 September 2021 | Accepted: 28 September 2021 | Published on: 21 October 2021

DOI: 10.1002/qj.4181

Quarterly Journal of the Royal Meteorological Society RMetS

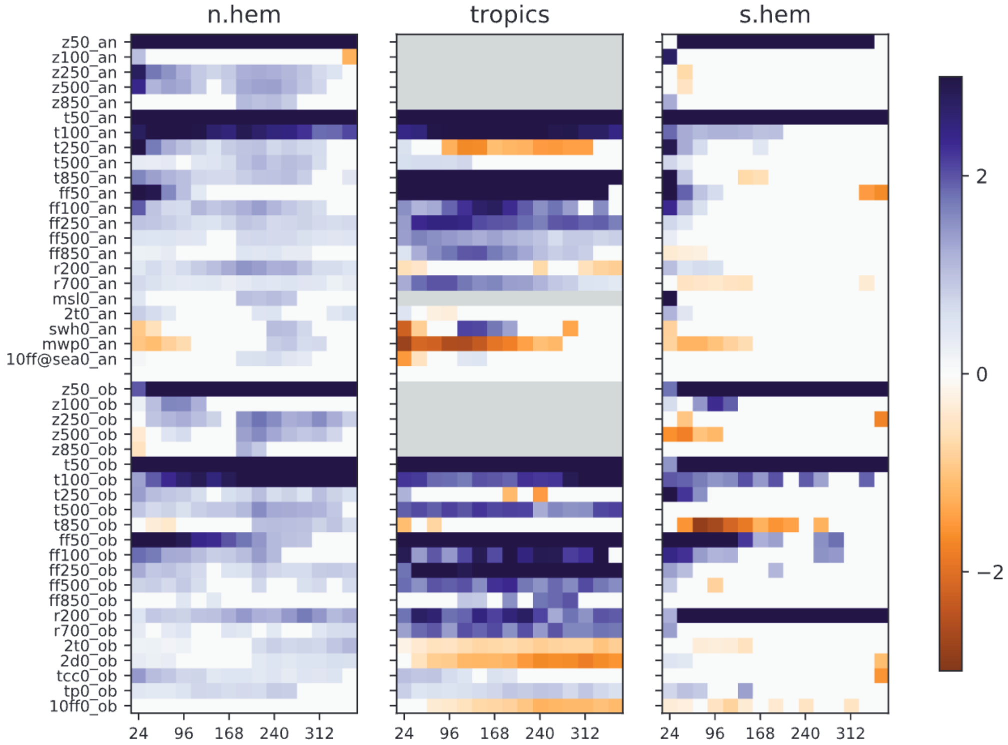
RESEARCH ARTICLE

More accuracy with less precision

Simon T. K. Lang[1] | **Andrew Dawson**[1] | **Michail Diamantakis**[1] |
Peter Dueben[1] | **Samuel Hatfield**[1] | **Martin Leutbecher**[1] | **Tim Palmer**[2] |
Fernando Prates[1] | **Christopher D. Roberts**[1] | **Irina Sandu**[1] | **Nils Wedi**[1]

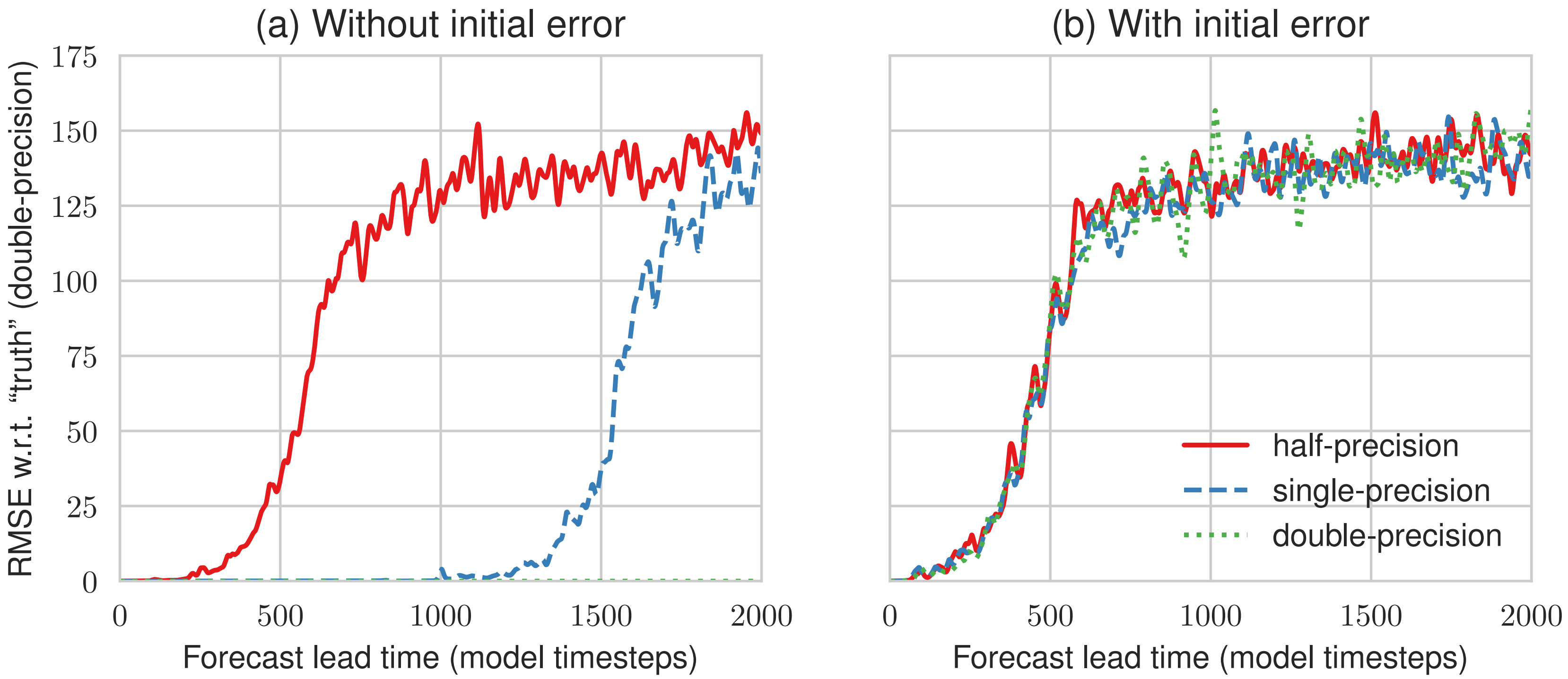
Lang et al. 2021

Switching from double to single precision permitted vertical resolution increase in ENS **for free**
→ **improves forecast skill**



Scorecard SP vs DP (+ve = SP better)

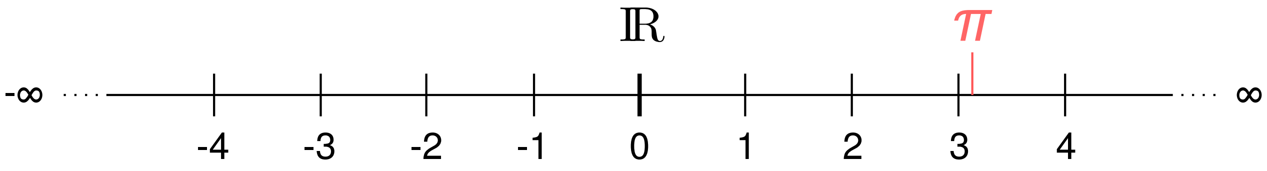
How much does precision matter?



Lorenz '63 example

Real numbers on computers

Numerical models use **real number arithmetic**



Computers deal with **finite bit strings**

$\pi \approx$

0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1
0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1 0 0 0 1 1 0 0 0

64 bits

How do we map a real number to a string of bits?

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0 = 182_{10}$$

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0 = 182_{10}$$

$$10110.110_2 = 1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0+1\times2^{-1}+1\times2^{-2}+0\times2^{-3} = 22.75_{10}$$

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0 = 182_{10}$$

$$10110.110_2 = 1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0+1\times2^{-1}+1\times2^{-2}+0\times2^{-3} = 22.75_{10}$$

Major drawback: limited range = $2^{\text{number of digits left of decimal place}} - 1$

A better way: floating-point numbers

Instead we use **floating-point numbers**:

$$x = \underbrace{\text{fixed-point number}}_{\substack{\text{significand/mantissa} \\ \text{(between 1 and 2)}}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

A better way: floating-point numbers

Instead we use **floating-point numbers**:

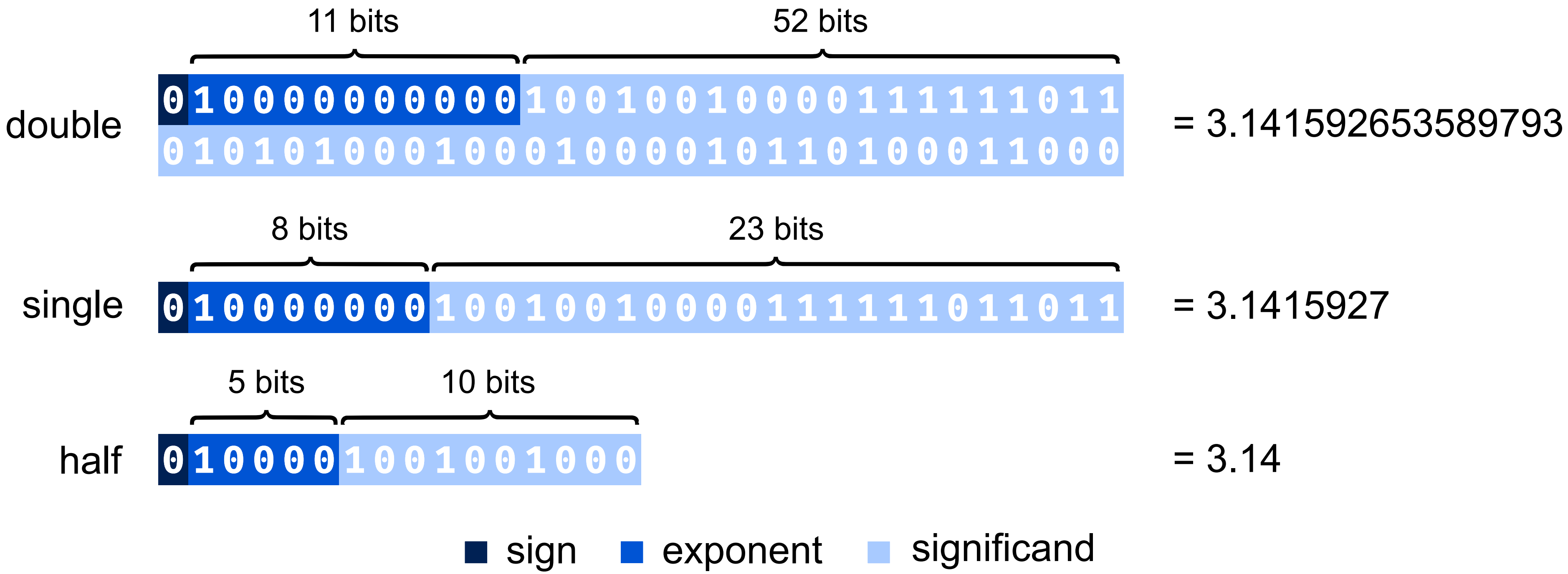
$$x = \underbrace{\text{fixed-point number}}_{\substack{\text{significand/mantissa}\\ \text{(between 1 and 2)}}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

$$10110110_2 = (1\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4}+0\times2^{-5})\times2^{1\times4+1\mathrm{x}2+0\mathrm{x}1-3} = 13.5_{10}$$

$$11111111_2 = (1\times2^0+1\times2^{-1}+1\times2^{-2}+1\times2^{-3}+1\times2^{-4}+1\times2^{-5})\times2^{1\times4+1\mathrm{x}2+1\mathrm{x}1-3} = 31.5_{10}$$

$$00000001_2 = (1\times2^0+0\times2^{-1}+0\times2^{-2}+0\times2^{-3}+0\times2^{-4}+0\times2^{-5})\times2^{0\times4+0\mathrm{x}2+1\mathrm{x}1-3} = 0.125_{10}$$

Boring but important: standardisation

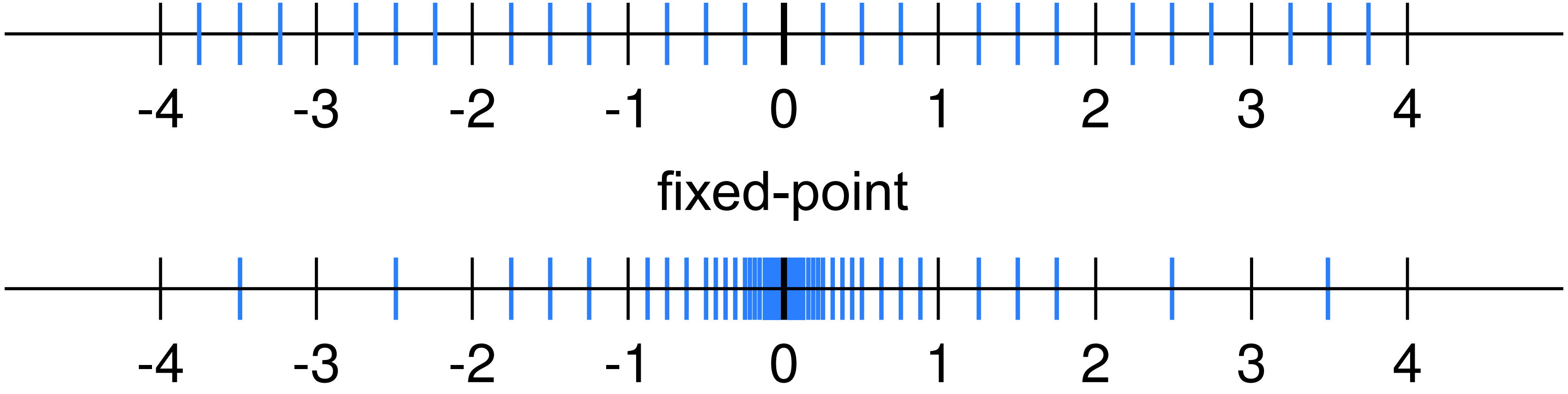


	sign	exponent	significand	
double	0	10000000000	10010010000111111011 01010100010001000010110100011000	= 3.141592653589793
single	0	10000000	10010010000111111011011	= 3.1415927
half	0	10000	1001001000	= 3.14

■ sign ■ exponent ■ significand

IEEE 754

Fixed- vs. floating-point number distribution

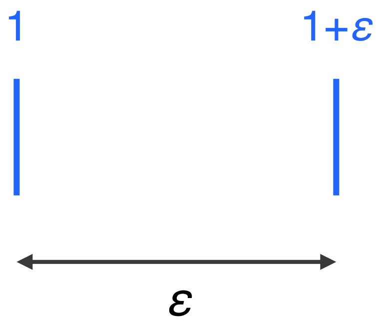


fixed-point

floating-point

Machine precision

The difference between 1 and the next largest representable number is called the *machine precision/epsilon*



$$\varepsilon = 2^{-\text{number of significand bits}}$$

The relative error of a floating-point assignment will be *at most ε* / 2

“Subnormal” numbers

Remember:

$$00000001_2 = (1\times2^{0}+0\times2^{-1}+0\times2^{-2}+0\times2^{-3}+0\times2^{-4}+0\times2^{-5})\times2^{0\times4+0x2+1x1-3} = 0.125_{10}$$

This system cannot represent zero because of this guy.

“Subnormal” numbers

Remember:

$$00000001_2 = (1\times2^{0}+0\times2^{-1}+0\times2^{-2}+0\times2^{-3}+0\times2^{-4}+0\times2^{-5})\times2^{0\times4+0\times2+1\times1-3} = 0.125_{10}$$

This system cannot represent zero because of this guy.

By convention, when the **exponent** is zero, the (implicit) leading bit is 0, not 1

$$00000000_2 = (0\times2^{0}+0\times2^{-1}+0\times2^{-2}+0\times2^{-3}+0\times2^{-4}+0\times2^{-5})\times2^{0\times4+0\times2+0\times1-3} = 0.0_{10}$$

Numbers with a zeroed **exponent** are called **subnormal numbers**

Flops

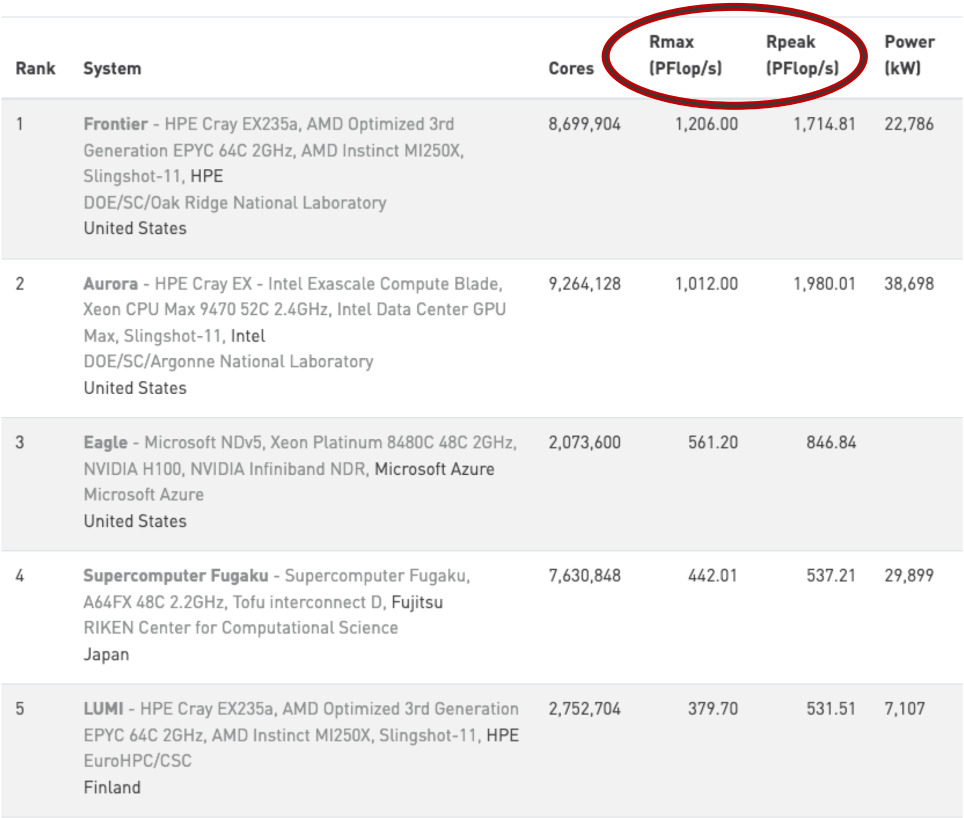
There are four elementary arithmetic operations:

+ − × ÷

Carrying out one of these **op**erations on two **fl**oating-point numbers constitutes one ***flop***

See also: sqrt and FMA

Flop/s: an imperfect measure



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

Top500 supercomputer rankings
June 2024

Rmax (PFlop/s)
Measured performance of LINPACK

Rpeak (PFlop/s)
Theoretical peak performance

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

`0.1` is actually `0.0999999`
`0.2` is actually `0.1999999`

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

`0.1` is actually `0.0999999`
`0.2` is actually `0.1999999`
`0.1 + 0.2` is actually `0.2998888`

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

`0.1` is actually `0.0999999`
`0.2` is actually `0.1999999`
`0.1 + 0.2` is actually `0.2998888`
`0.3` is actually `0.3000488`

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3
false # WTF?
```

What’s going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

`0.1` is actually `0.0999999`
`0.2` is actually `0.1999999`
`0.1 + 0.2` is actually `0.2998888`
`0.3` is actually `0.3000488`
hence
`0.1 + 0.2 != 0.3`

Floating-point numbers are weird: demonstration 2

The harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots$$

diverges when calculated with infinite precision

With (finite-precision) floating-point arithmetic, it converges!

Float16

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots = 7.0859$$

Float32

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots = 15.404$$

Float64

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots = 34.122$$

Swamping

The harmonic series converges because of *swamping*

```
julia> Float16(2500.0) + Float16(1.0)
Float16(2500.0)
```

This can occur when doing big number + small number. Why?

```
julia> nextfloat(Float16(2500.0))
Float16(2502.0) # There is no Float16(2501.0)!
```

Floating-point numbers are weird: demonstration 3

Pathological case:

*a, b are arrays of **50,000** elements of all **1**s except the first which is **50***

Answer with Float32:
`52499.0`

Answer with Float16:
`2500.0`

```
function inner_product(a, b)
    sum = a[1] * b[1]

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        sum += a_i * b_i
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Pathological case:

*a, b are arrays of **50,000** elements of all **1**s except the first which is **50***

Answer with Float32:
`52499.0`

Answer with Float16:
`52499.0`

```
function inner_product_mixed(a, b)
    sum = Float32(a[1] * b[1])

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        sum += Float32(a_i * b_i)
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Pathological case:

a, b are arrays of ***50,000*** *elements of all* ***1****s except the first which is* ***50***

Answer with Float32:
`52499.0`

Answer with Float16:
`52500.0`

```
function inner_product_compensated(a, b)
    sum = a[1] * b[1]
    c = convert(typeof(a[1]), 0.0)

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        y = (a_i * b_i) - c
        t = sum + y
        c = (t - sum) - y
        sum = t
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Base type	Algorithm	Answer	FLOPs
Float32	Basic	`52499.0`	2n-1 Float32
Float16	Basic	`2500.0`	2n-1 Float16
Float16	Mixed	`52499.0`	n Float16 + n-1 Float32
Float16	Compensated	`52500.0`	5n-4 Float16

Floating-point numbers: *recap*

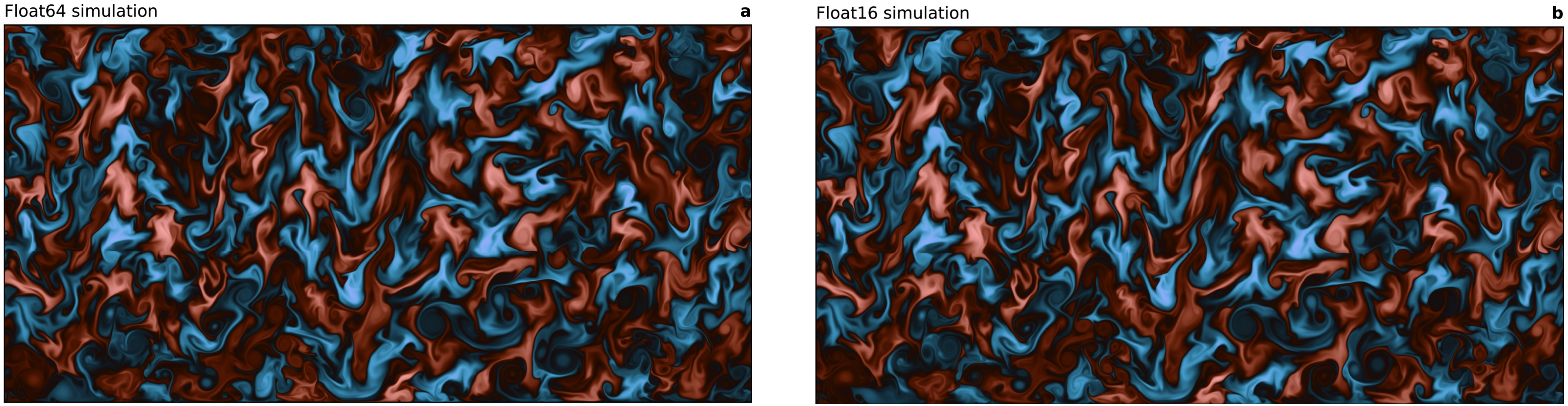
- Floating-point numbers have a **significand** and an **exponent**

$$x = \underbrace{\text{fixed-point number}}_{\text{significand}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

- Their precision is determined by the **machine epsilon**
- They have a **normal** range and a **subnormal** range
- “Computational work” is measured in **flops**, speed in **flop/s**
- Only numbers decomposable into power-two sums are perfectly representable
- Caution required when adding **big numbers** and **small numbers**

Half-precision in practice

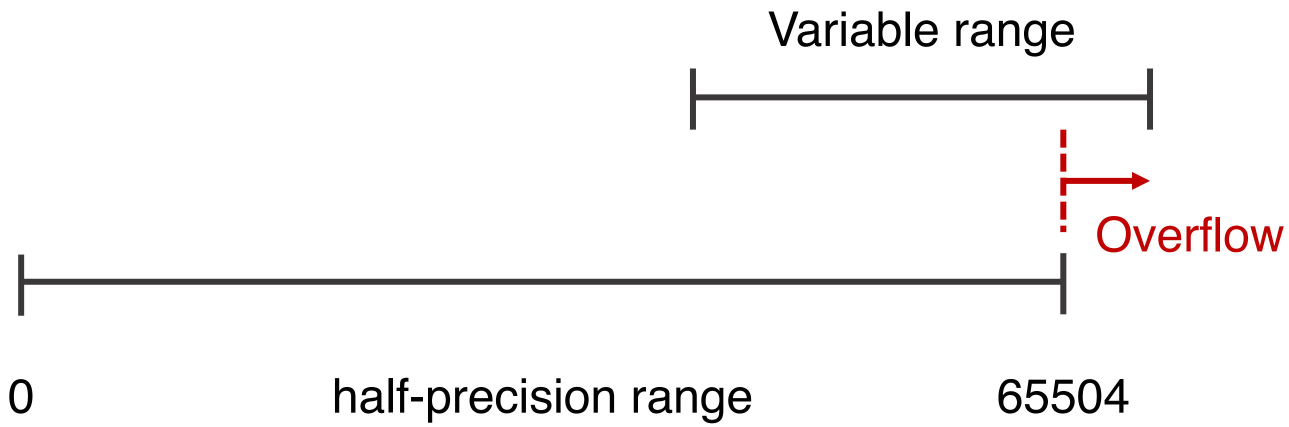
Half precision already demonstrated in shallow water simulations



Klöwer et al. 2021

What about in NWP models?

Could we use half precision in NWP?



Variable range

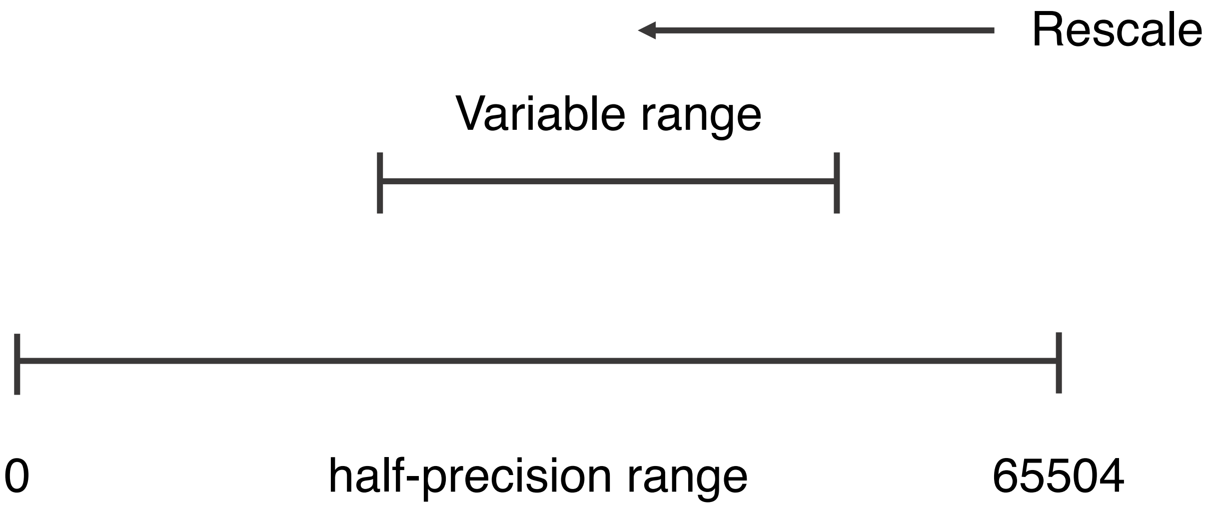
Overflow

0

half-precision range

65504

Could we use half precision in NWP?



Rescale

Variable range

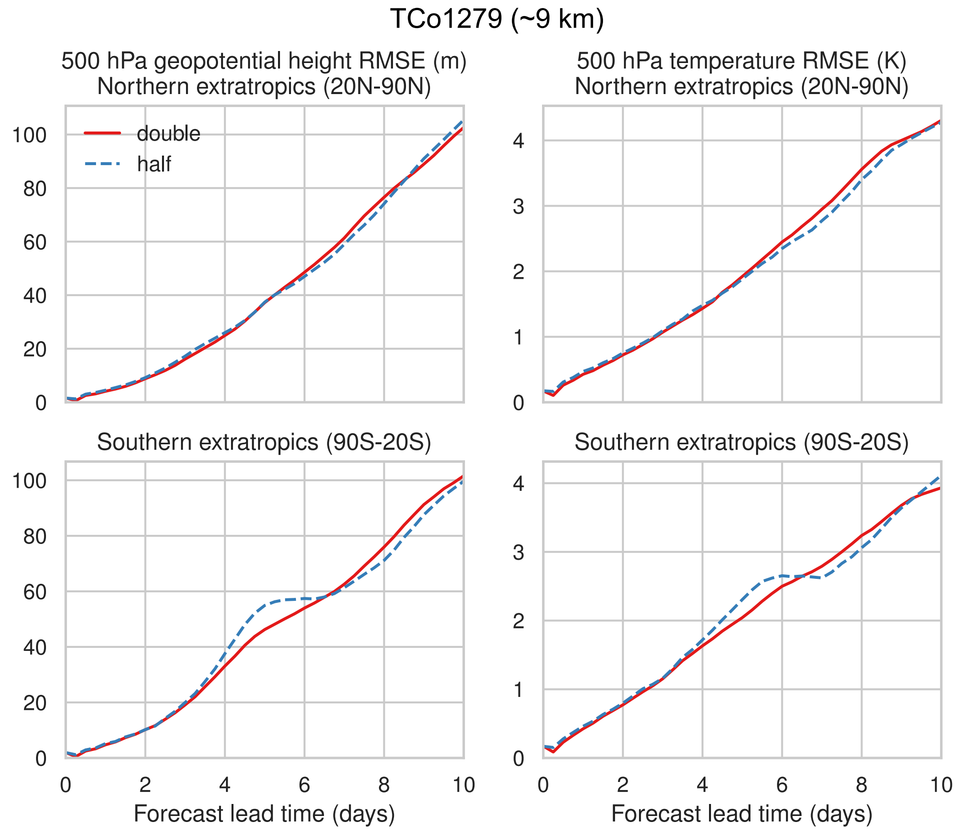
0

half-precision range

65504

Could we use half precision in NWP?

It actually works…
(for software-emulated half precision)



Hatfield et al. 2019

What about data assimilation?

Successful convergence of minimization in 4D-Var depends on this equality holding:

$$[\underbrace{\mathbf{M}_{im}}_{\text{tangent-linear model}}(\mathbf{x}_0)\boldsymbol{\delta}\mathbf{x}]^{\mathrm{T}}\boldsymbol{\delta}\mathbf{y} = \boldsymbol{\delta}\mathbf{x}^{\mathrm{T}}[\underbrace{\mathbf{M}_{im}^{\mathrm{T}}}_{\text{adjoint model}}(\mathbf{x}_0)\boldsymbol{\delta}\mathbf{y}]$$

This equality will never hold exactly for floating-point arithmetic!
The higher the precision, the less the inequality

See Hatfield et al. 2020

Sam Hatfield

samuel.hatfield@ecmwf.int