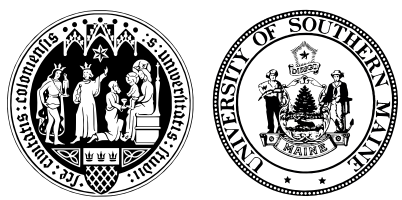
Evaluation of Bfloat16, Posit, and Takum Arithmetics in Sparse Linear Solvers

Laslo Hunhold[1] James Quinlan[2]

[1]Parallel and Distributed Systems Group, University of Cologne, Germany

[2]Department of Computer Science, University of Southern Maine, Portland, ME, USA

5th May 2025



Introduction

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range $\rightarrow$ `bfloat16`

Introduction

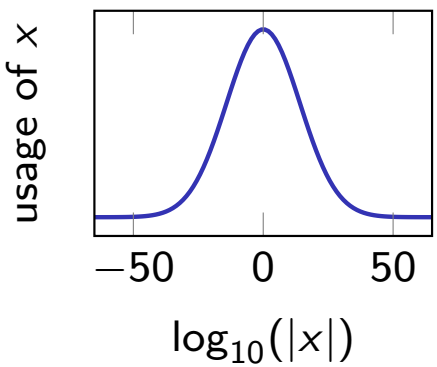
- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range $\rightarrow$ `bfloat16`
  - Number distribution does not reflect real-world usage

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range $\rightarrow$ `bfloat16`
  - Number distribution does not reflect real-world usage $\rightarrow$ tapered precision arithmetic

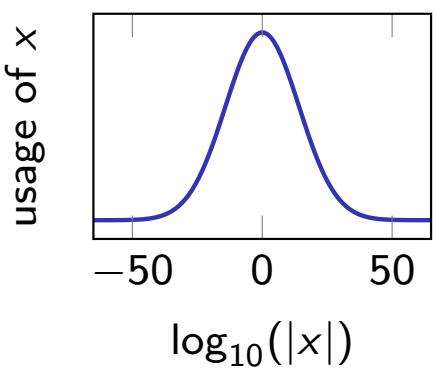
Introduction

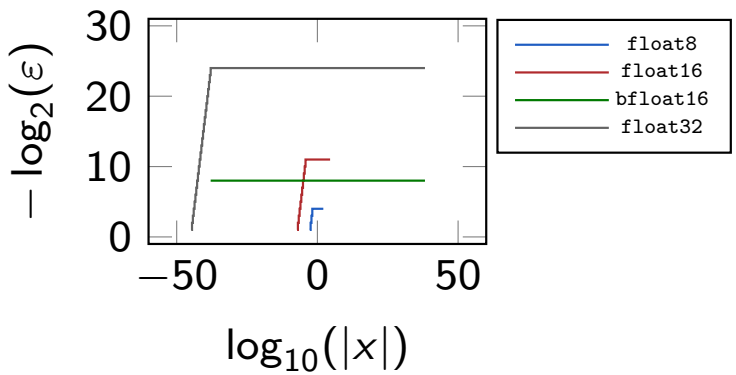
- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type $\rightarrow$ OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range $\rightarrow$ IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range $\rightarrow$ `bfloat16`
  - Number distribution does not reflect real-world usage $\rightarrow$ tapered precision arithmetic



Introduction

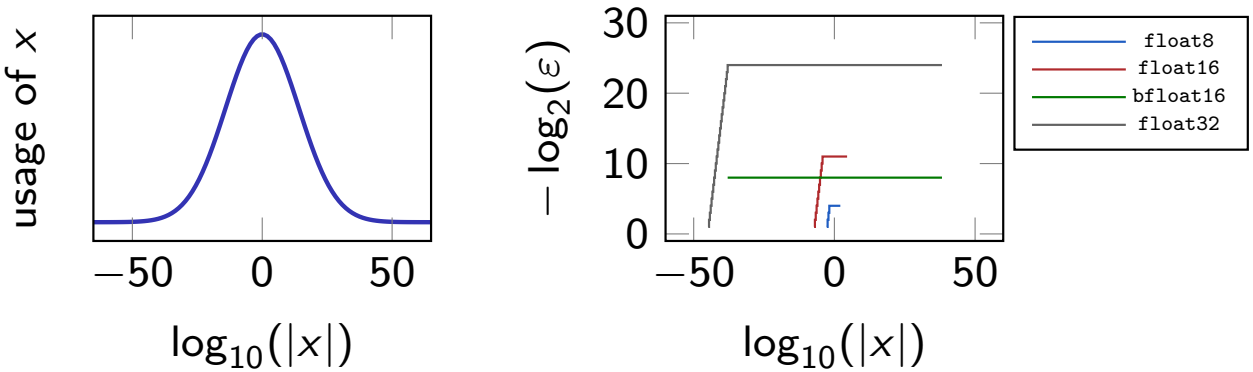
- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type → OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range → IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range → `bfloat16`
  - Number distribution does not reflect real-world usage → tapered precision arithmetic





Introduction

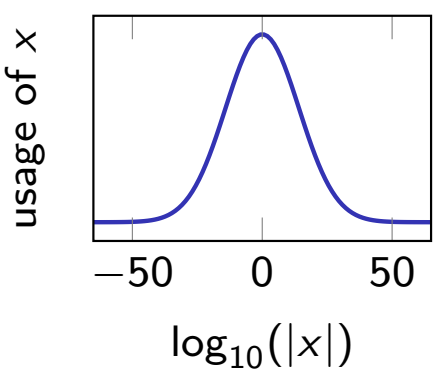
- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type → OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range → IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range → `bfloat16`
  - Number distribution does not reflect real-world usage → tapered precision arithmetic

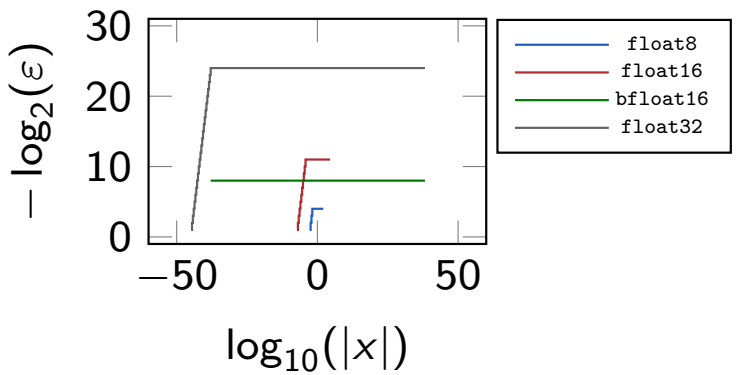


- Strong focus on deep learning → Performance in general-purpose computing?

Introduction

- Memory wall dictates move to smaller machine number types (end of 'doubles everywhere')
- Overparametrization of DNNs makes low-precision arithmetic sufficient
- Common criticisms of IEEE 754 for low-precision arithmetic
  - No standardized 8-bit type → OCP OFP8 (*E4M3*, E5M2)
  - Redundancies dominate at low precisions, overflows due to small dynamic range → IEEE P3109 (one NaN/zero, saturation)
  - `float16` has insufficient dynamic range → `bfloat16`
  - Number distribution does not reflect real-world usage → tapered precision arithmetic



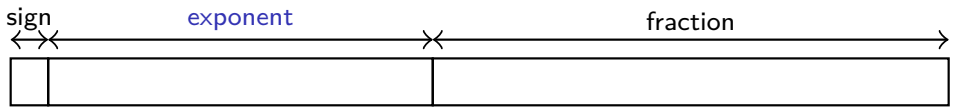


- Strong focus on deep learning → Performance in general-purpose computing?
- This work: Evaluation within sparse linear solvers

Tapered Precision Arithmetic

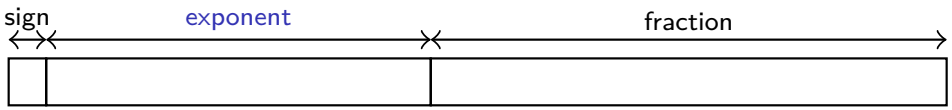
Tapered Precision Arithmetic

- Consider general floating-point format



Tapered Precision Arithmetic

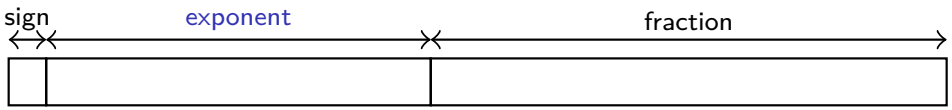
- Consider general floating-point format



- Sign and fraction bits already contain maximum information

Tapered Precision Arithmetic

- Consider general floating-point format

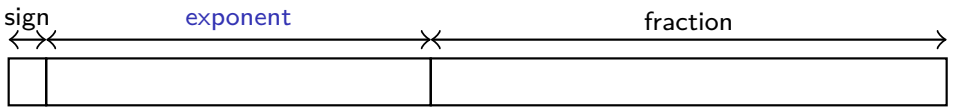


- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

Tapered Precision Arithmetic

- Consider general floating-point format



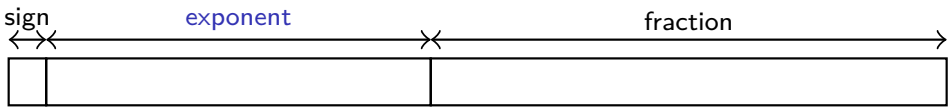
- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, precision) for exponent values close to zero

Tapered Precision Arithmetic

- Consider general floating-point format



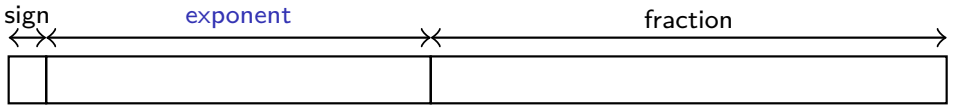
- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, precision) for exponent values close to zero
- Solution: variable-length exponent encoding

Tapered Precision Arithmetic

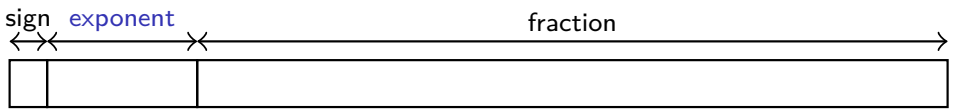
- Consider general floating-point format



- Sign and fraction bits already contain maximum information

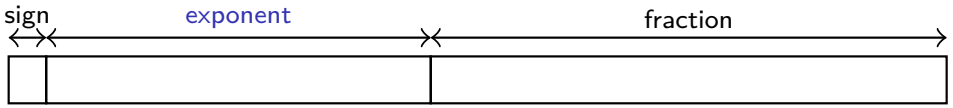
Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, precision) for exponent values close to zero
- Solution: variable-length exponent encoding
  - Small magnitude: shorter exponent, longer fraction, higher density



Tapered Precision Arithmetic

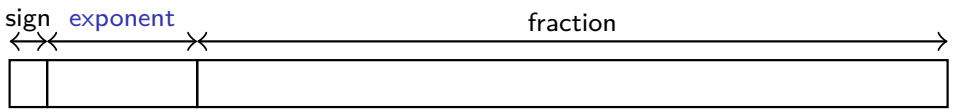
- Consider general floating-point format



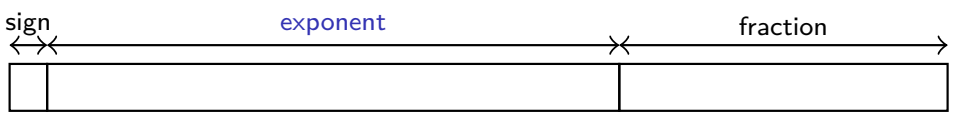
- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, precision) for exponent values close to zero
- Solution: variable-length exponent encoding
  - Small magnitude: shorter exponent, longer fraction, higher density



  - Large magnitude: longer exponent, shorter fraction, lower density



Posit Arithmetic

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

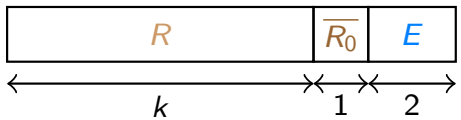
- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding

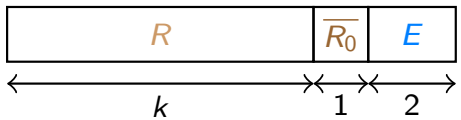


[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



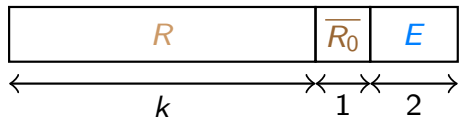
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



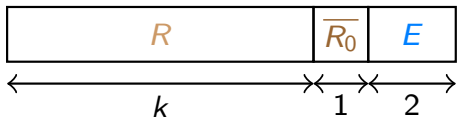
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



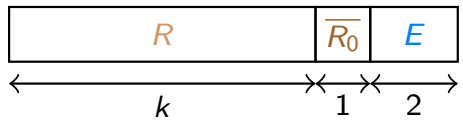
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



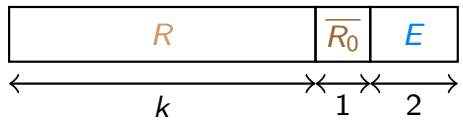
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



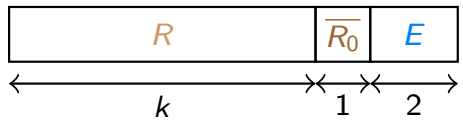
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



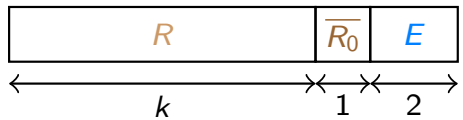
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



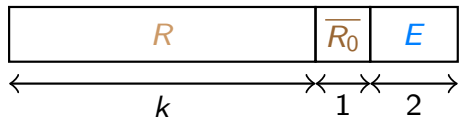
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



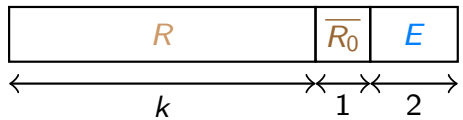
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



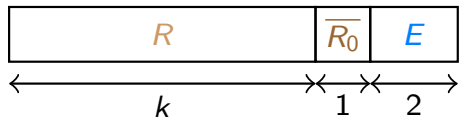
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



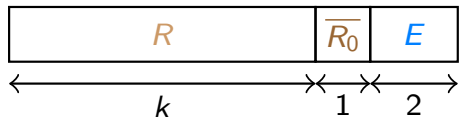
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



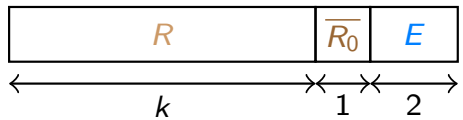
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



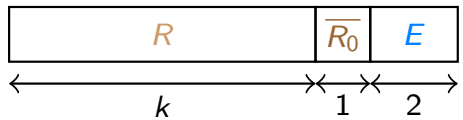
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering, sign)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

Posit Arithmetic

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



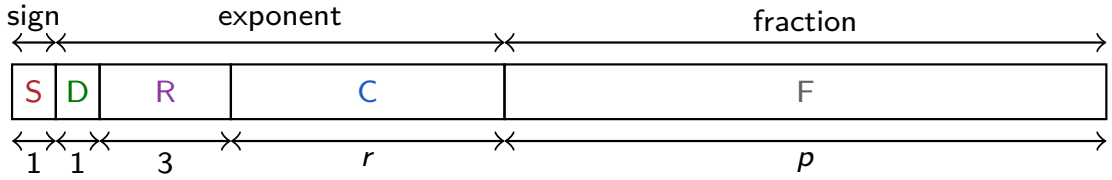
  - R is run of k zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)[2]
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering, sign)
  - Defined for all n, conversion between lengths simple rounding or expansion

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

[2] Florent de Dinechin et al. 'Posits: the good, the bad and the ugly'. Mar. 2019

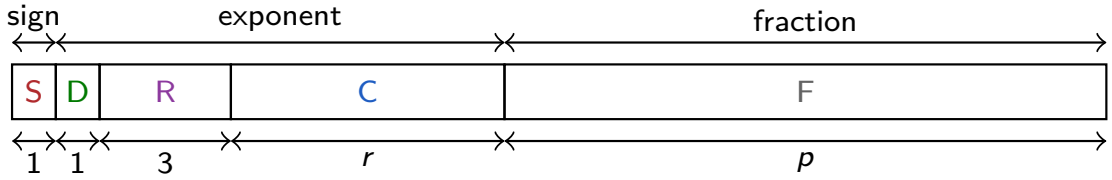
Takum Arithmetic

Takum Arithmetic

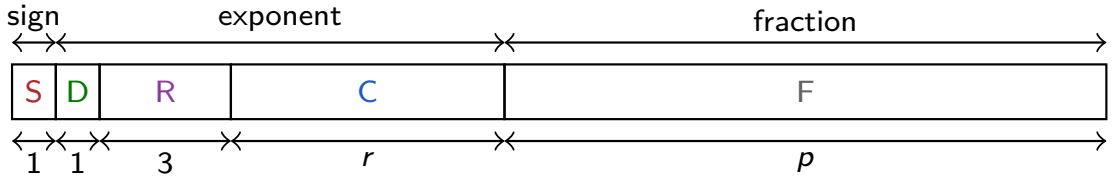


Takum Arithmetic

- Goals

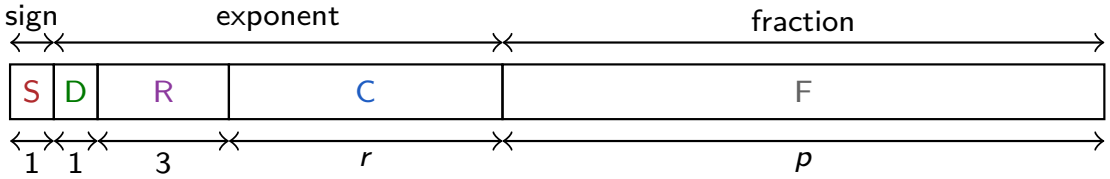


Takum Arithmetic



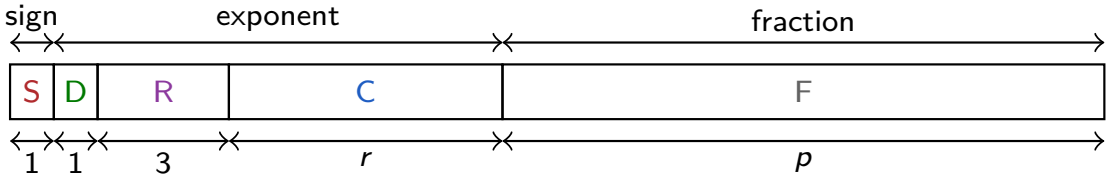
- Goals
  - More efficient exponent code

Takum Arithmetic



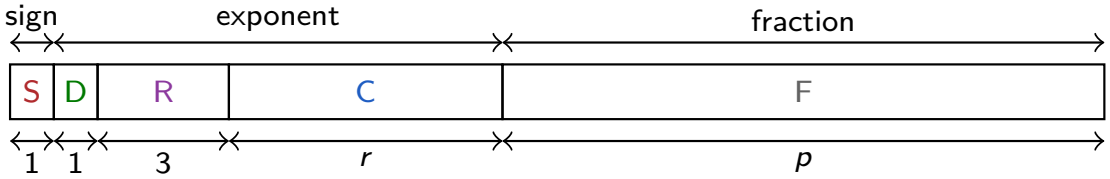
- Goals
  - More efficient exponent code
  - Preserve useful posit properties

Takum Arithmetic



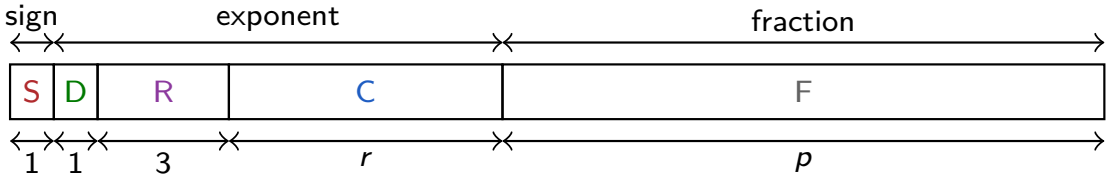
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach

Takum Arithmetic



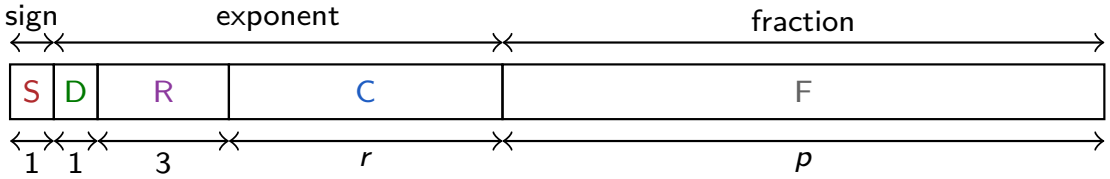
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value

Takum Arithmetic



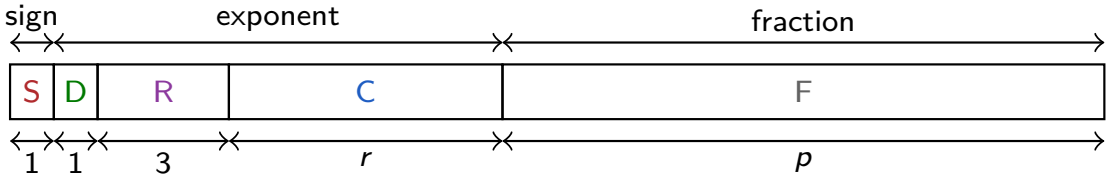
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated

Takum Arithmetic



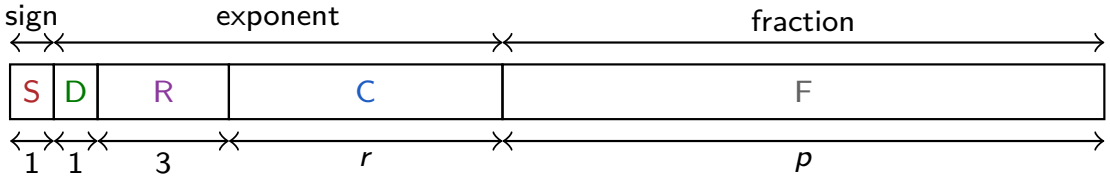
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$

Takum Arithmetic



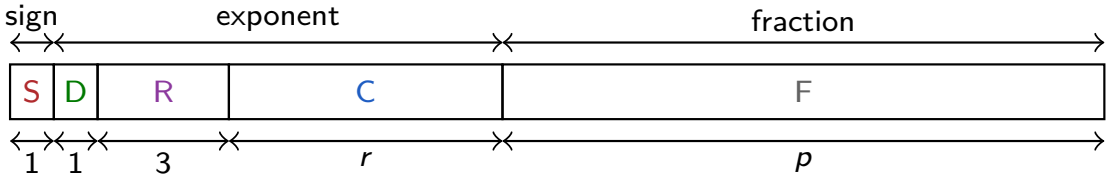
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally

Takum Arithmetic



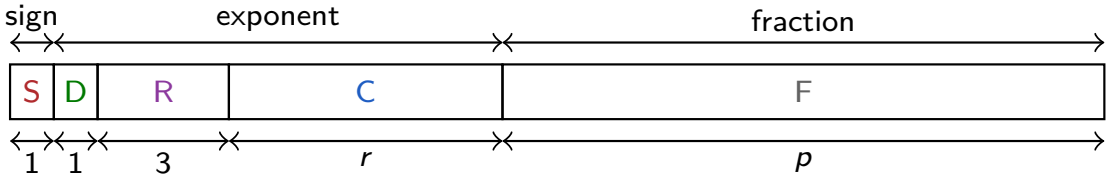
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.

Takum Arithmetic



- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

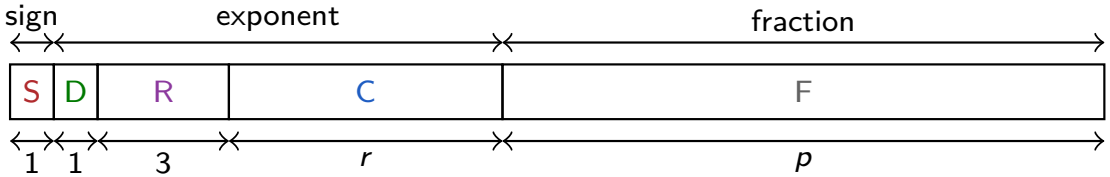
Takum Arithmetic



- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

Takum Arithmetic

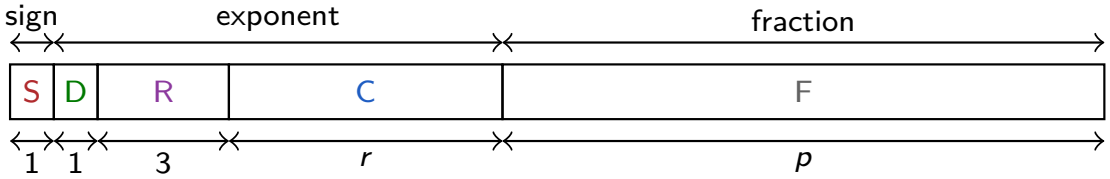


- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'

Takum Arithmetic

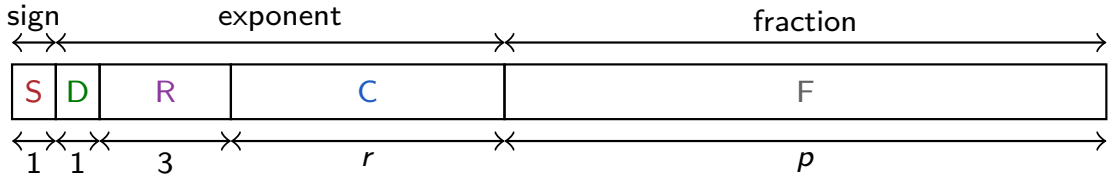


- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign

Takum Arithmetic

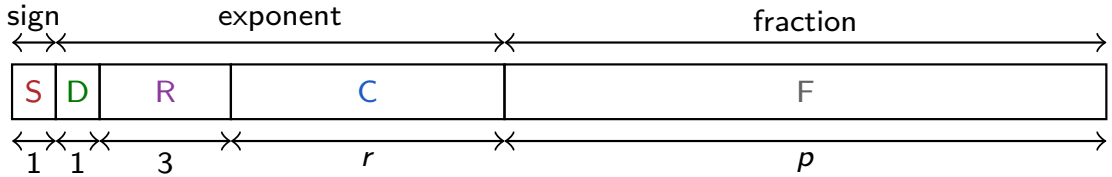


- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit → takum):

Takum Arithmetic



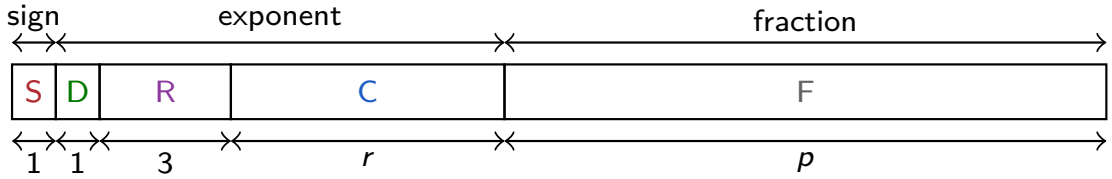
- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit → takum):

11111111111111111111111111111111111111111111111111111111111111010 (67 bit)

Takum Arithmetic



- Goals
  - More efficient exponent code
  - Preserve useful posit properties
- Design Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

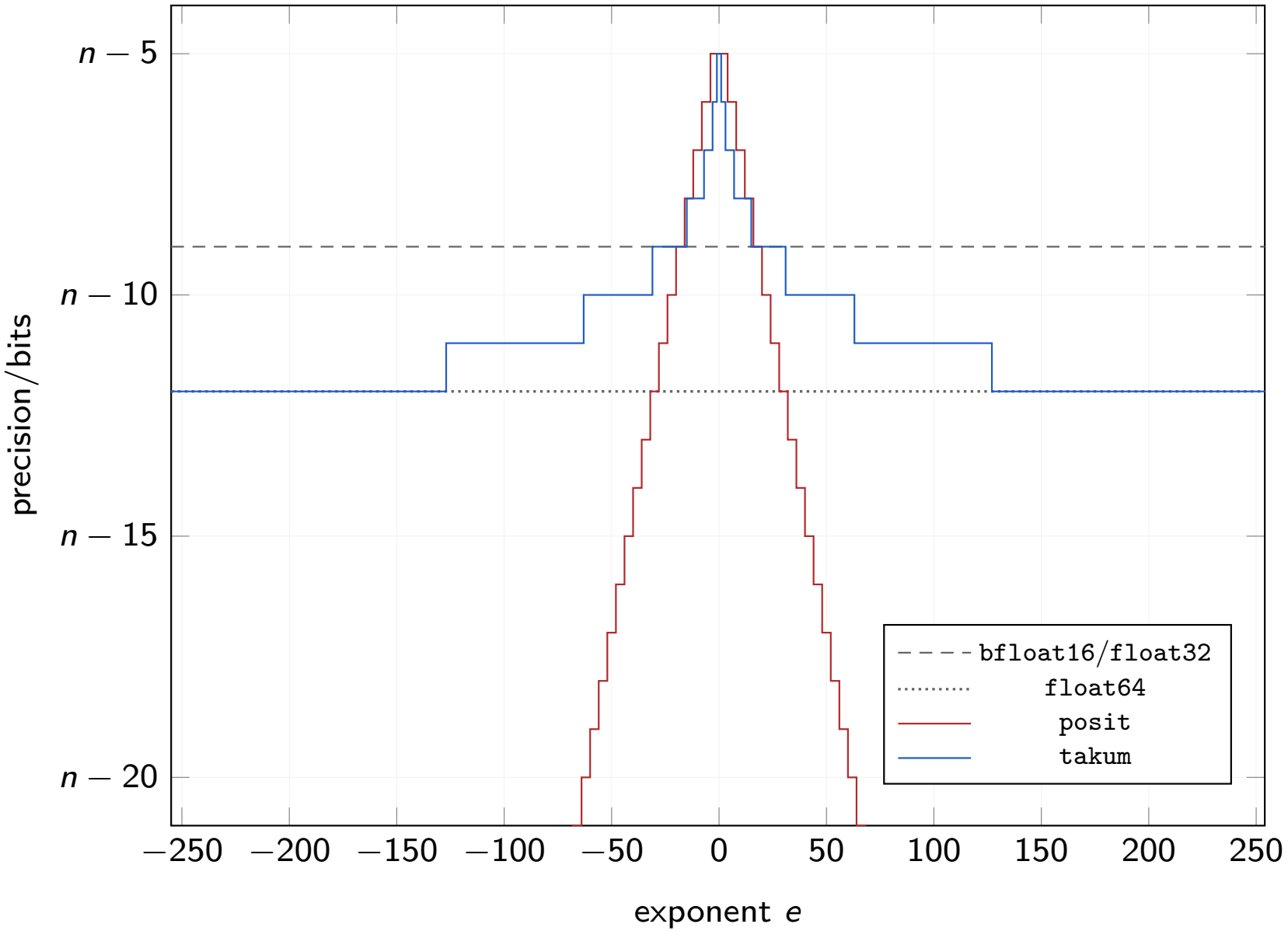
- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit → takum):

11111111111111111111111111111111111111111111111111111111111111010 (67 bit)

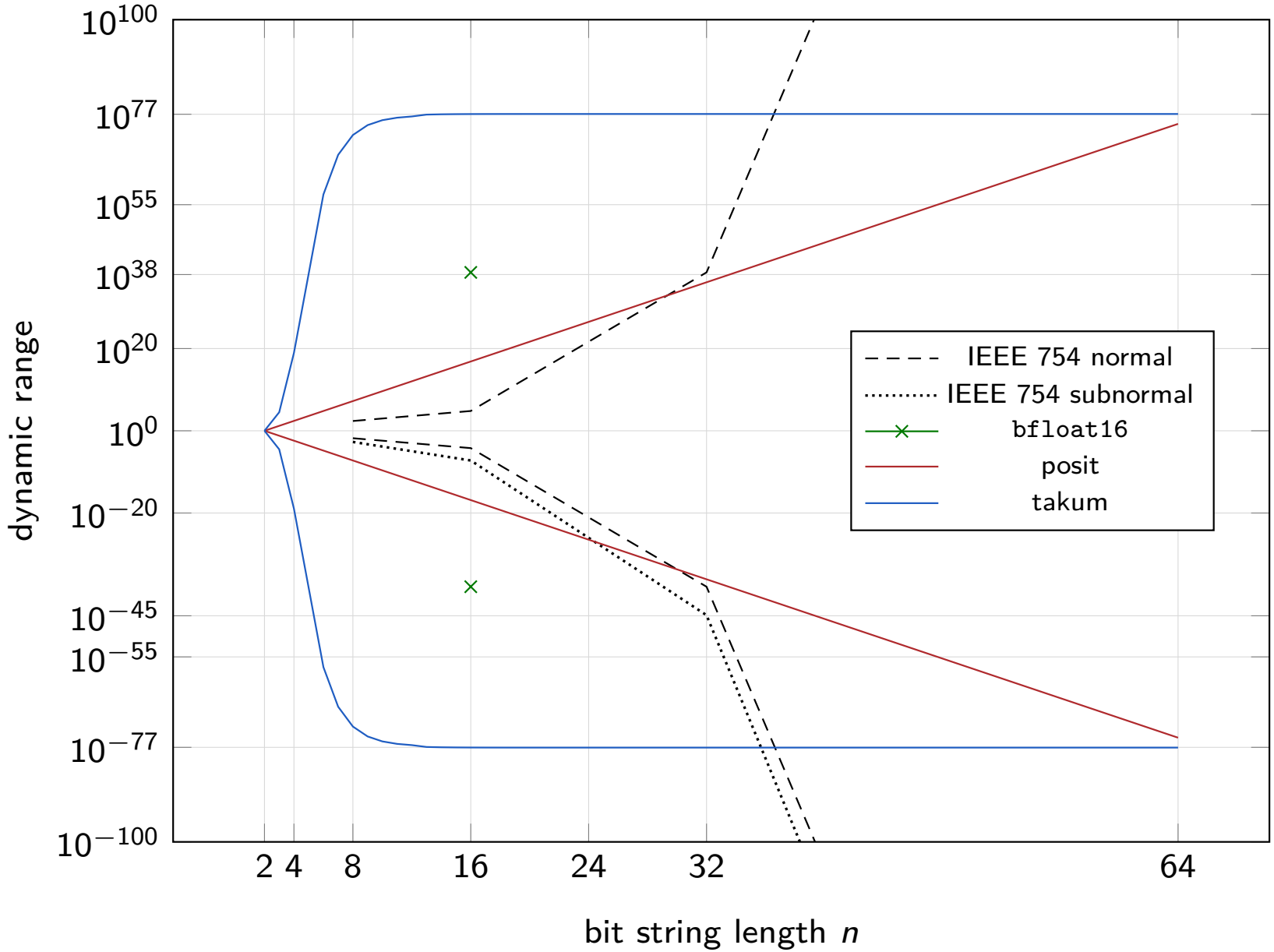
↓

11111111111 (11 bit)

Precision


$n-5$
$n-10$
$n-15$
$n-20$
precision/bits
−250
−200
−150
−100
−50
0
50
100
150
200
250
exponent e
bfloat16/float32
float64
posit
takum

Dynamic Range



Experimental Setup

Experimental Setup

Four solvers

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)
2. Solve system $Ax = b$ in `float128` using custom sparse QR solver

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)
2. Solve system $Ax = b$ in `float128` using custom sparse QR solver
3. Convert (A, b) to T, yielding $(\tilde{A}, \tilde{b})$

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)
2. Solve system $Ax = b$ in `float128` using custom sparse QR solver
3. Convert (A, b) to T, yielding $(\tilde{A}, \tilde{b})$
4. Solve $\tilde{A}\tilde{x} = \tilde{b}$

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)
2. Solve system $Ax = b$ in `float128` using custom sparse QR solver
3. Convert (A, b) to T, yielding $(\tilde{A}, \tilde{b})$
4. Solve $\tilde{A}\tilde{x} = \tilde{b}$
5. Determine $\|x - \tilde{x}\|_2$ in `float128`

Experimental Setup

Four solvers

- LU decomposition (simulate UMFPACK $\rightarrow$ pregenerated permutations)
- QR factorization (simulate SPQR $\rightarrow$ pregenerated permutations)
- Mixed Precision Iterative Refinement (MPIR) with ILU(0) preconditioning
- Generalized Minimal Residual (GMRES) method

Dataset

- SuiteSparse Matrix Collection
- Subset of 295 square, full rank matrices with less than 10^4 non-zero entries
- Custom metadata postprocessing and packing

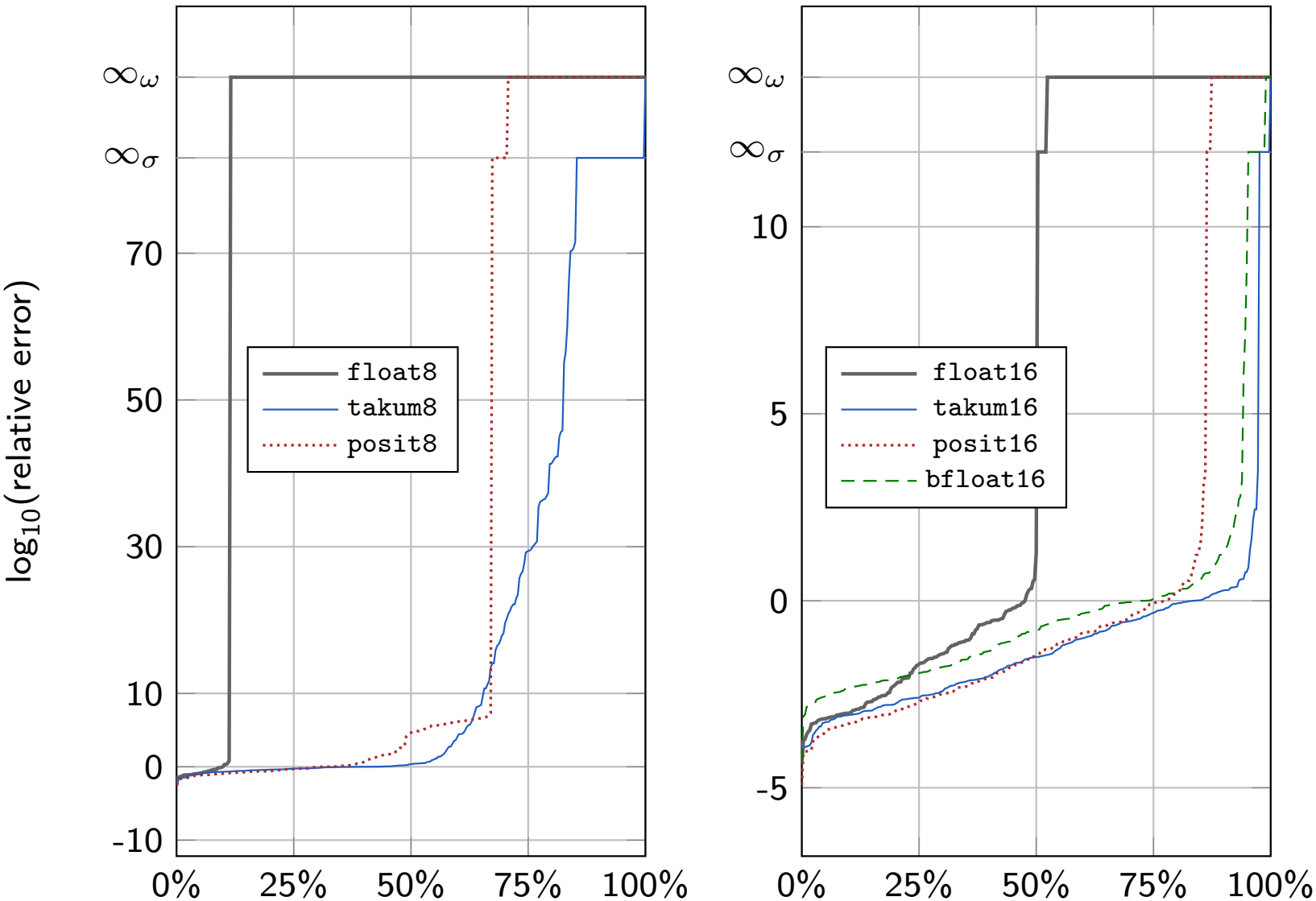
Procedure for each matrix A and type T

1. Generate random b such that $\|b\|_\infty = 1$ (seeded Xoshiro PRNG)
2. Solve system $Ax = b$ in `float128` using custom sparse QR solver
3. Convert (A, b) to T, yielding $(\tilde{A}, \tilde{b})$
4. Solve $\tilde{A}\tilde{x} = \tilde{b}$
5. Determine $\|x - \tilde{x}\|_2$ in `float128`

Sort all errors to obtain cumulative error distribution

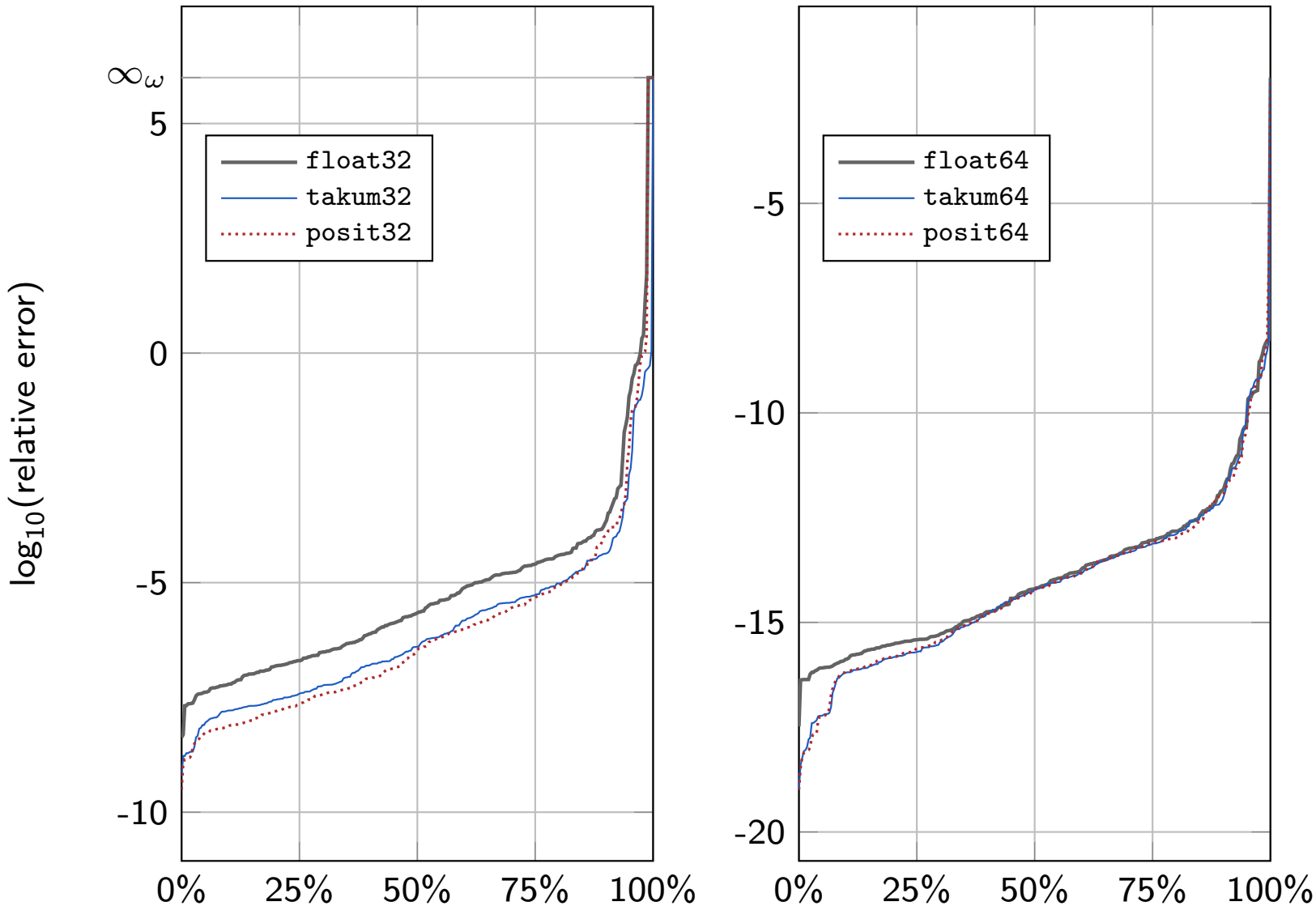
LU (1/2)

8 and 16 bits



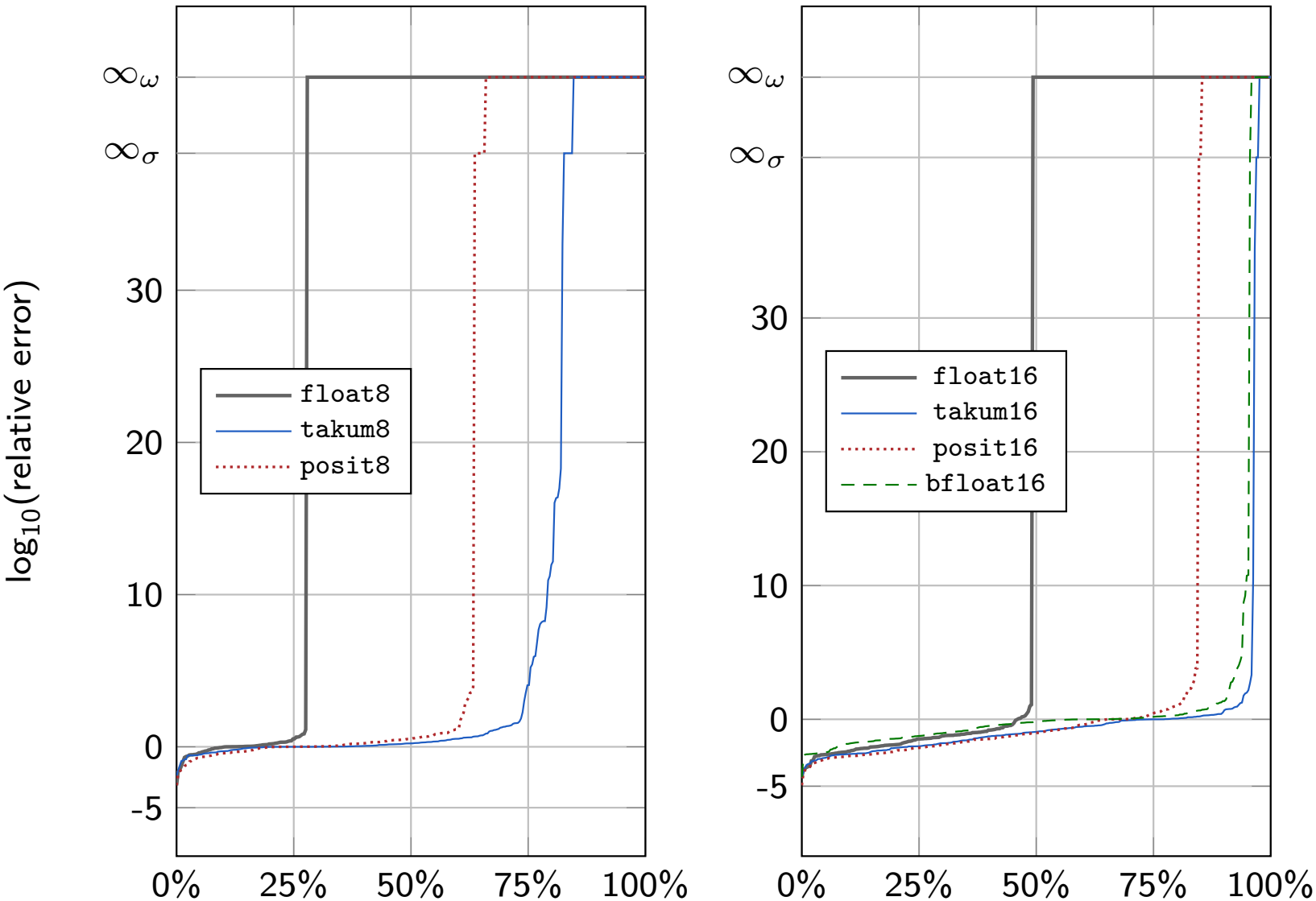
LU (2/2)

32 and 64 bits



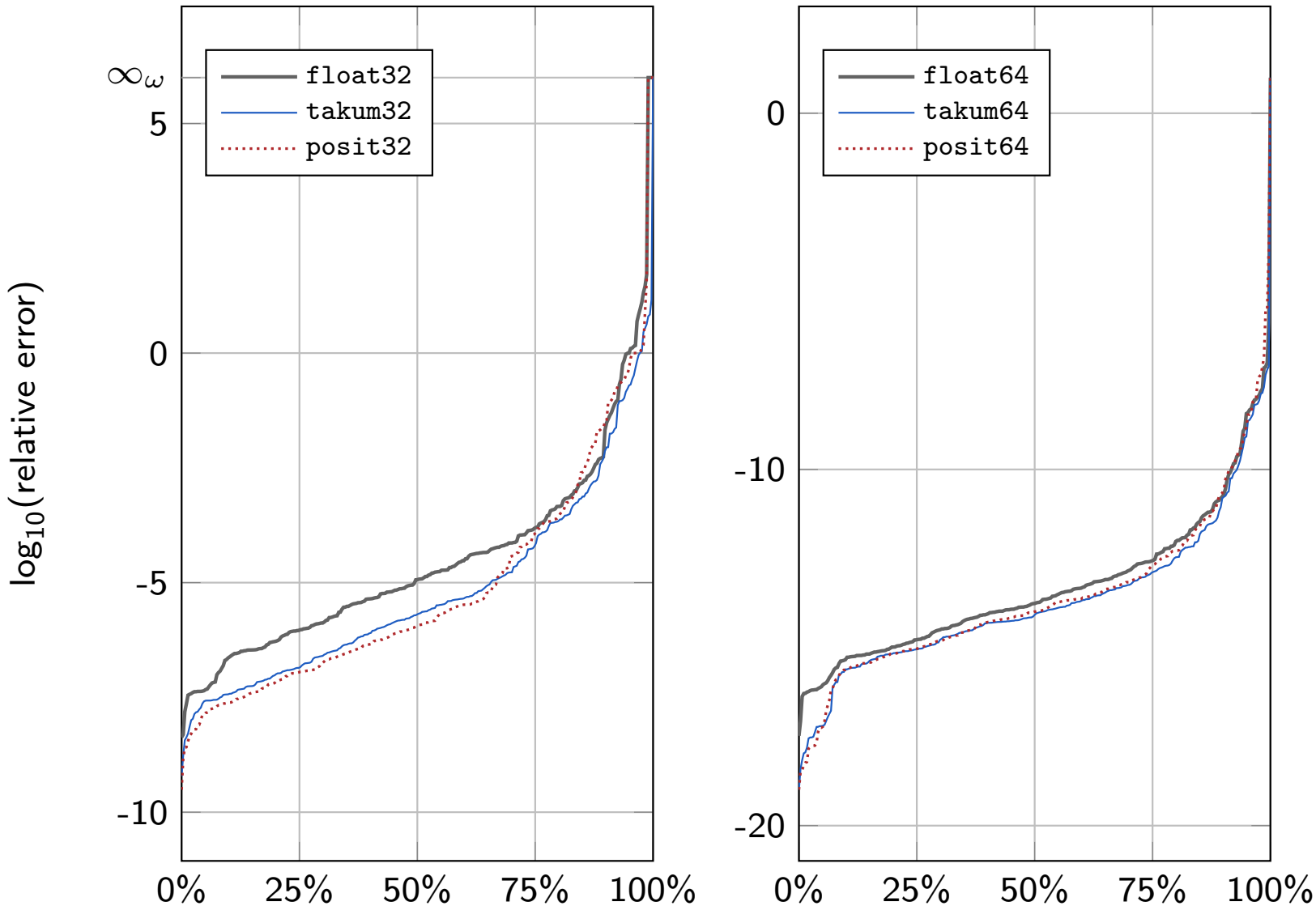
QR (1/2)

8 and 16 bits



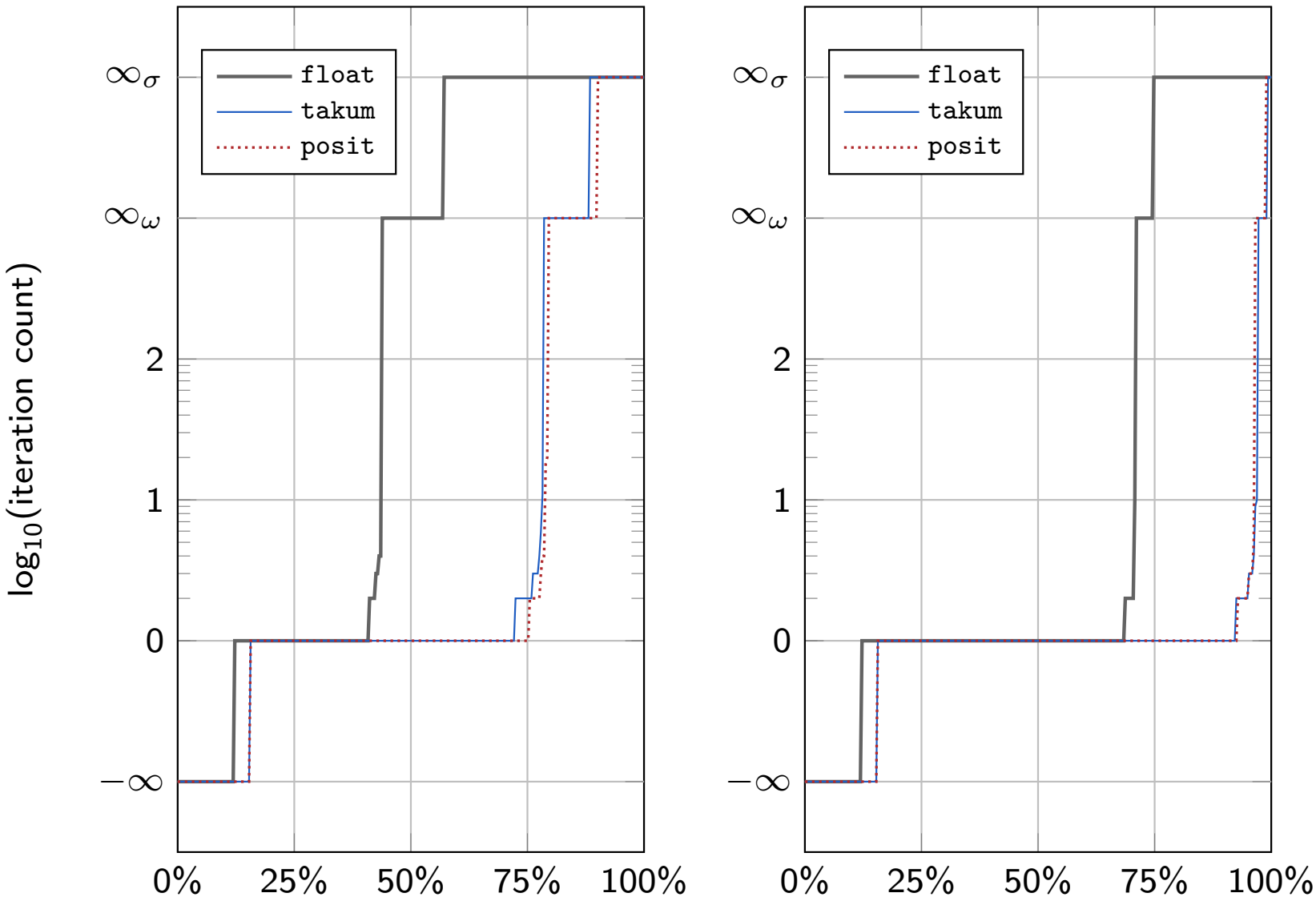
QR (2/2)

32 and 64 bits



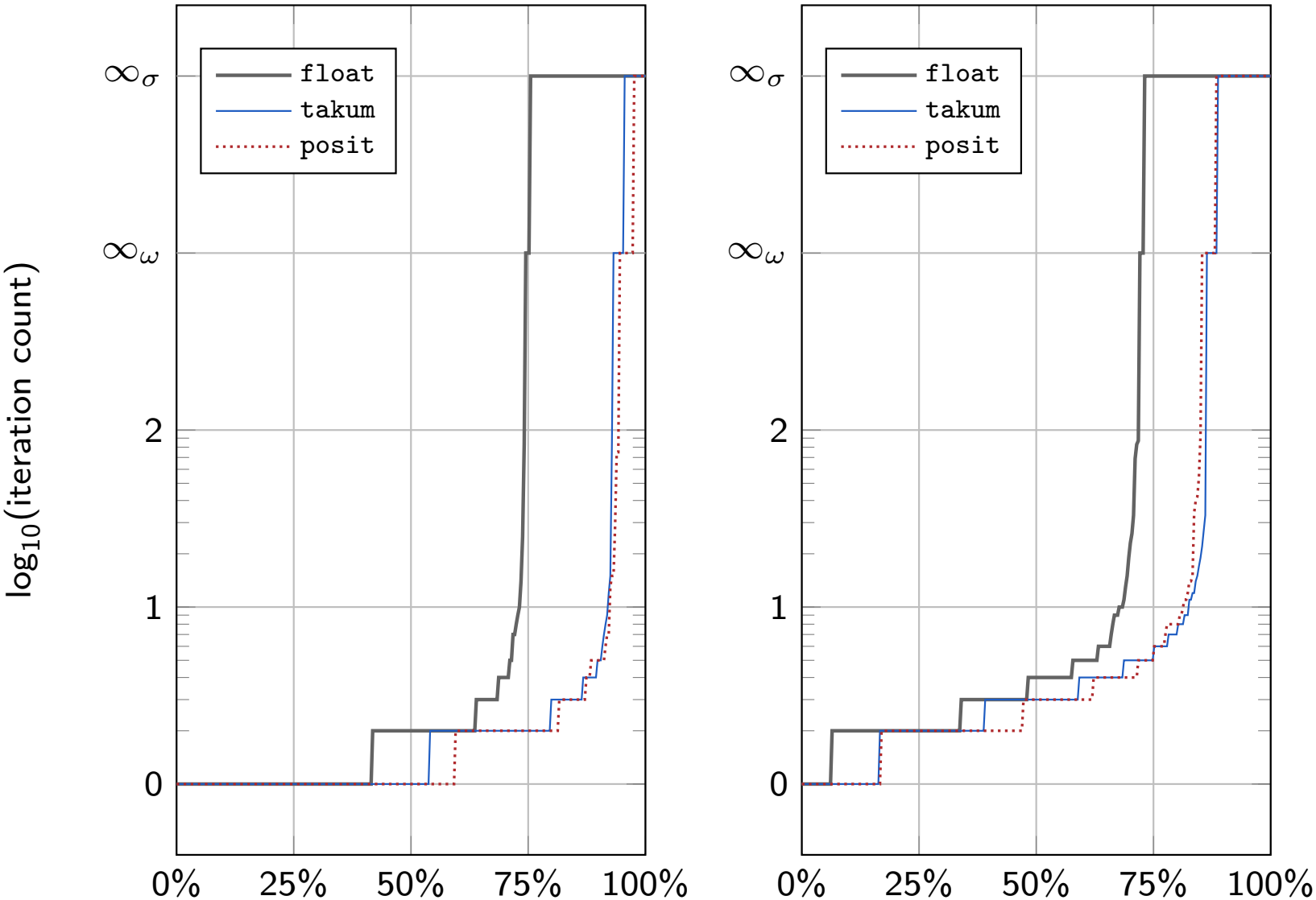
MPIR (1/2)

$(L, W, H) = (8, 16, 32)$ and $(L, W, H) = (16, 16, 32)$, relative tolerance 10^{-3} and 10^{-3}



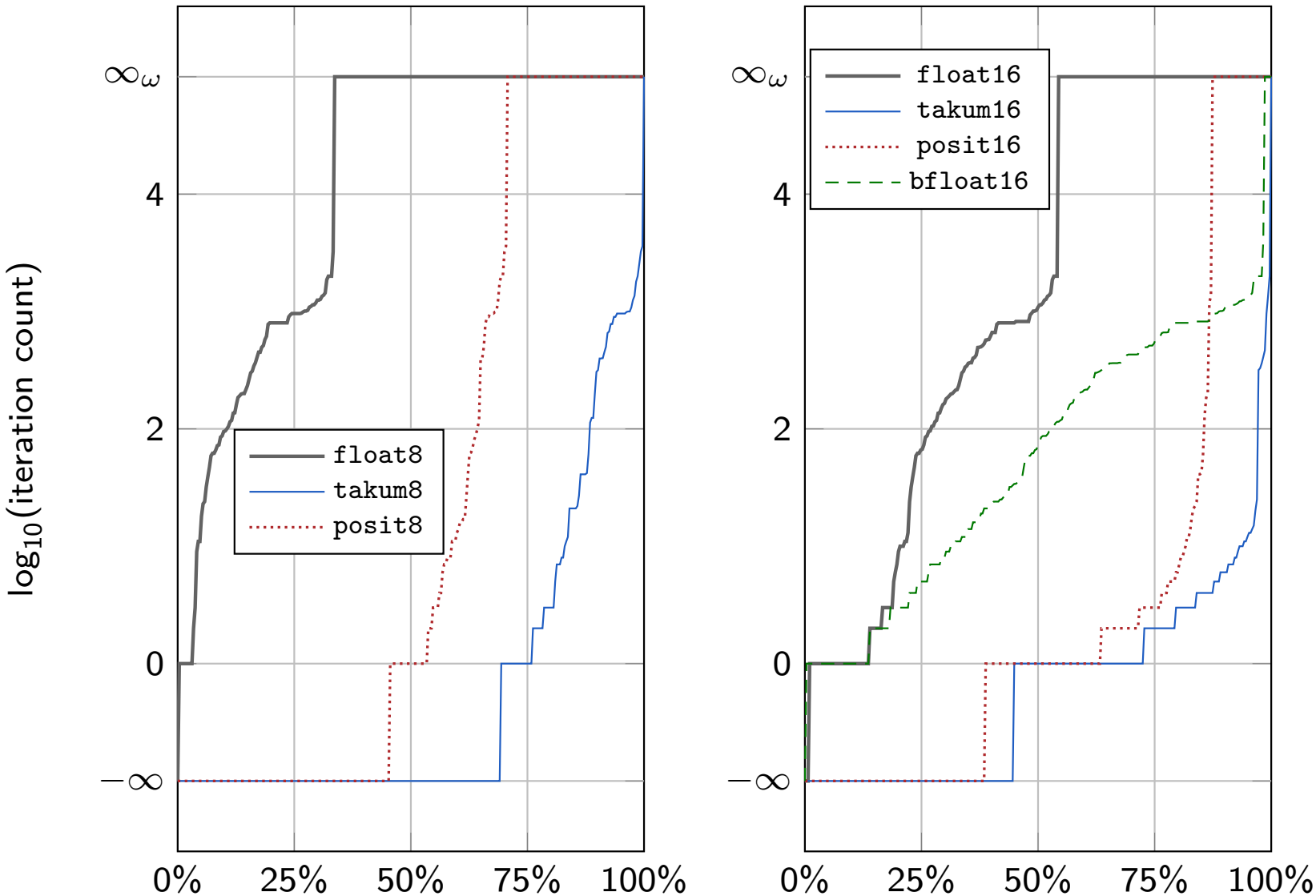
MPIR (2/2)

$(L, W, H) = (16, 32, 32)$ and $(L, W, H) = (16, 32, 64)$, relative tolerance 10^{-6} and 10^{-9}



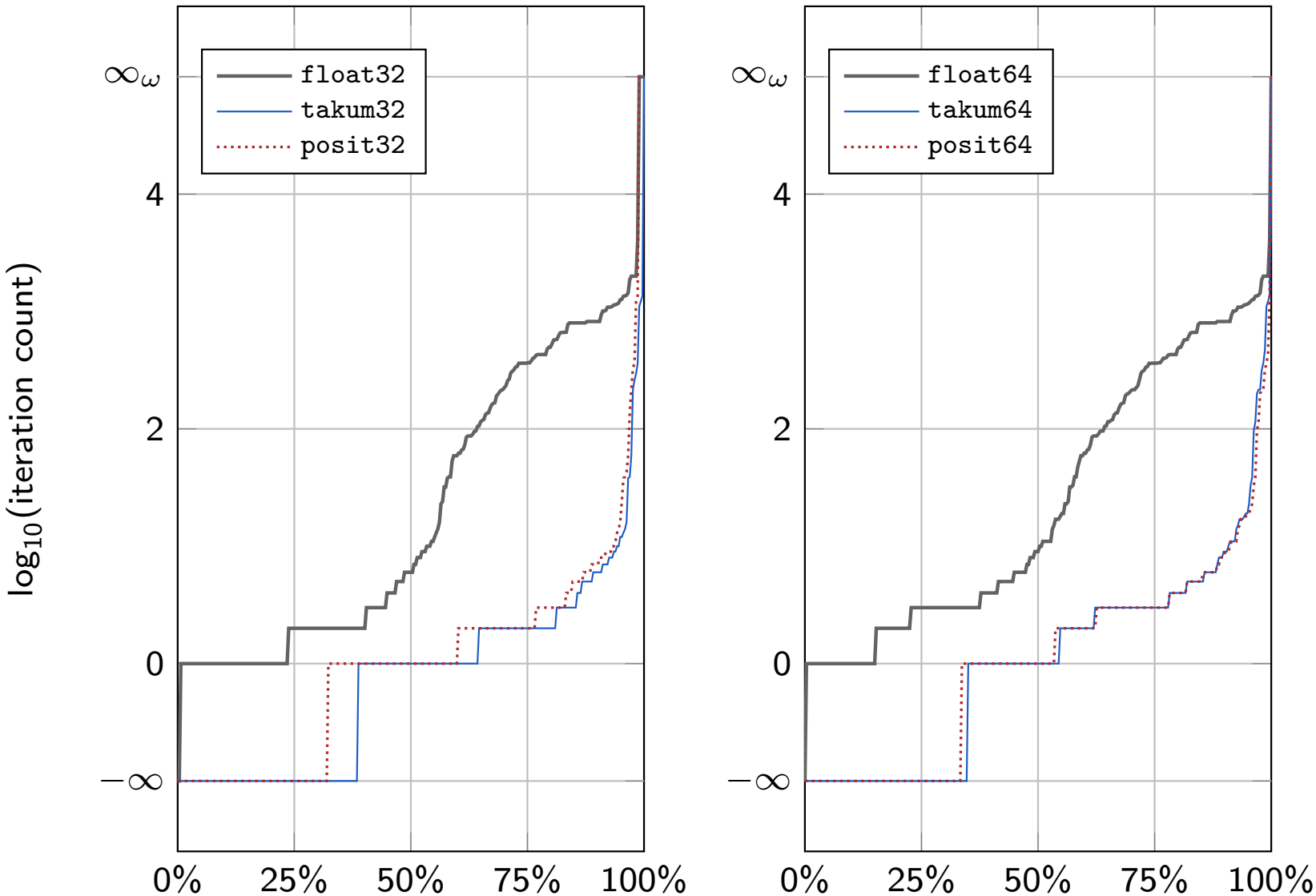
GMRES (1/2)

8 and 16 bits, restart value 20, relative tolerance $\sqrt{\varepsilon_{\texttt{float8}}}$ and $\sqrt{\varepsilon_{\texttt{float16}}}$



GMRES (2/2)

32 and 64 bits, restart value 20, relative tolerance $\sqrt{\varepsilon_{\texttt{float32}}}$ and $\sqrt{\varepsilon_{\texttt{float64}}}$



Conclusion

Conclusion

Summary of Results

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)
- Takums significantly outperform posits in some benchmarks, especially GMRES (against intuition)

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)
- Takums significantly outperform posits in some benchmarks, especially GMRES (against intuition)

Discussion

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)
- Takums significantly outperform posits in some benchmarks, especially GMRES (against intuition)

Discussion

- `bfloat16` is a better general-purpose format than `float16`

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)
- Takums significantly outperform posits in some benchmarks, especially GMRES (against intuition)

Discussion

- `bfloat16` is a better general-purpose format than `float16`
- Posits have some shortcomings (precision loss further from 1, limited dynamic range), but overall better than IEEE 754 floats

Conclusion

Summary of Results

- `bfloat16` overall better than `float16`, but sometimes worse
- Posits and takums overall superior than IEEE 754 floats
- `takum16` always better than `bfloat16` (unlike `posit16`)
- Takums significantly outperform posits in some benchmarks, especially GMRES (against intuition)

Discussion

- `bfloat16` is a better general-purpose format than `float16`
- Posits have some shortcomings (precision loss further from 1, limited dynamic range), but overall better than IEEE 754 floats
- Takums suggest new mixed-precision workflow: Reducing n only affects precision, not dynamic range