A composite-precision format for multiprecision matrix computations

SC25 St. Louis, Nov 16-21, 2025

Randomized Numerical Linear Algebra in HPC: Toward a Sustainable, Scalable Software Ecosystem

Vasileios Georgiou, Purdue
Petros Drineas, Purdue

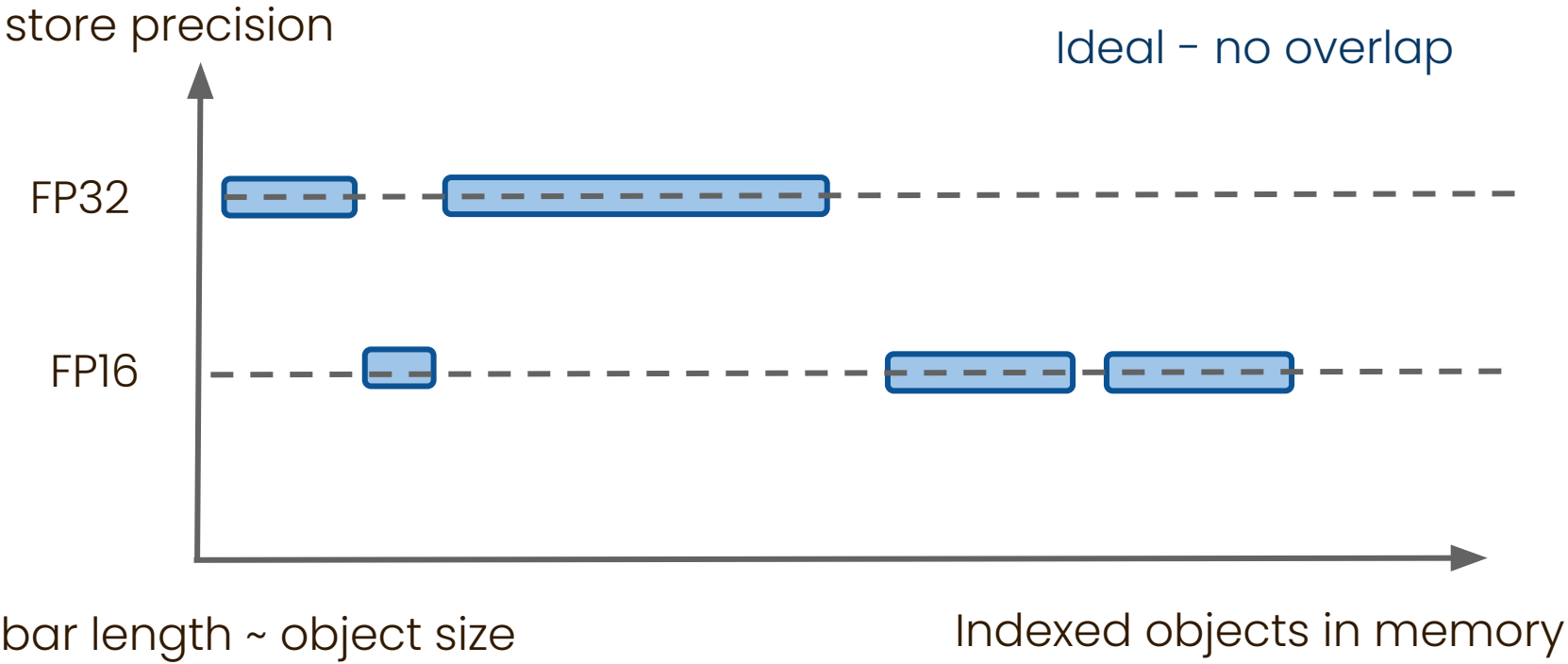
Motivation

New accelerators provide better and better “mixed-precision performance”

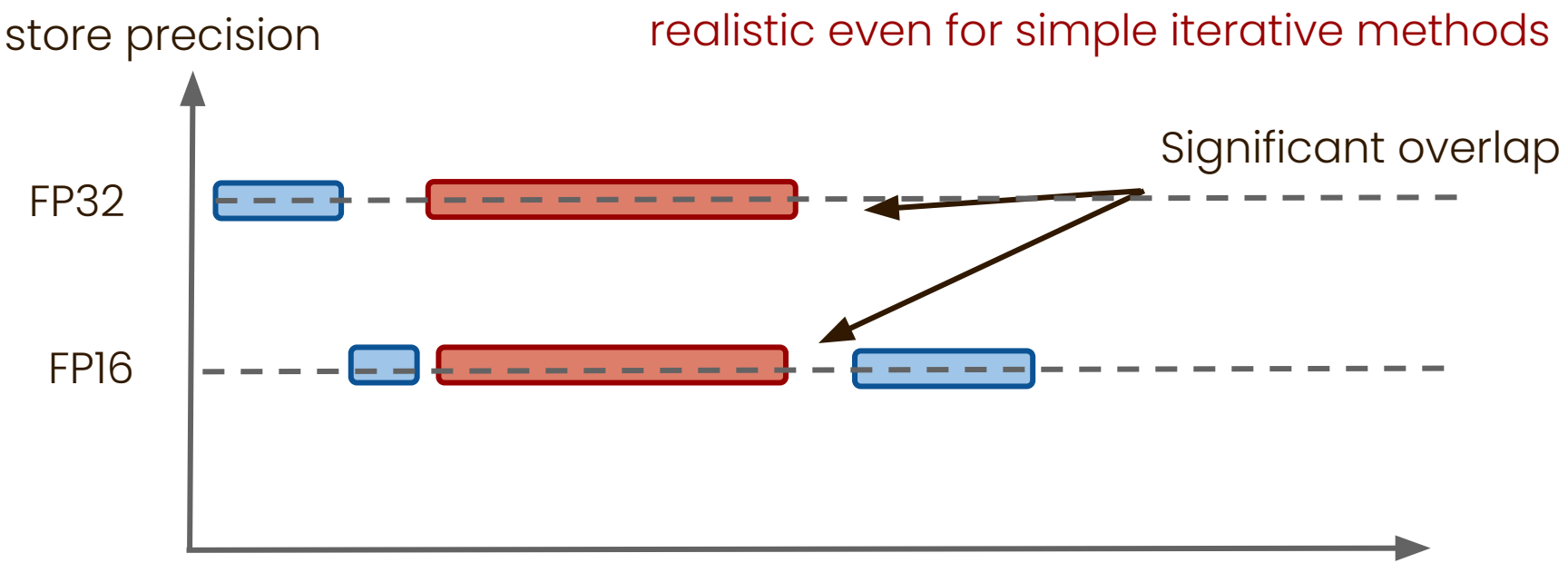
Simple idea:

- reduce memory transfers
- accelerate compute

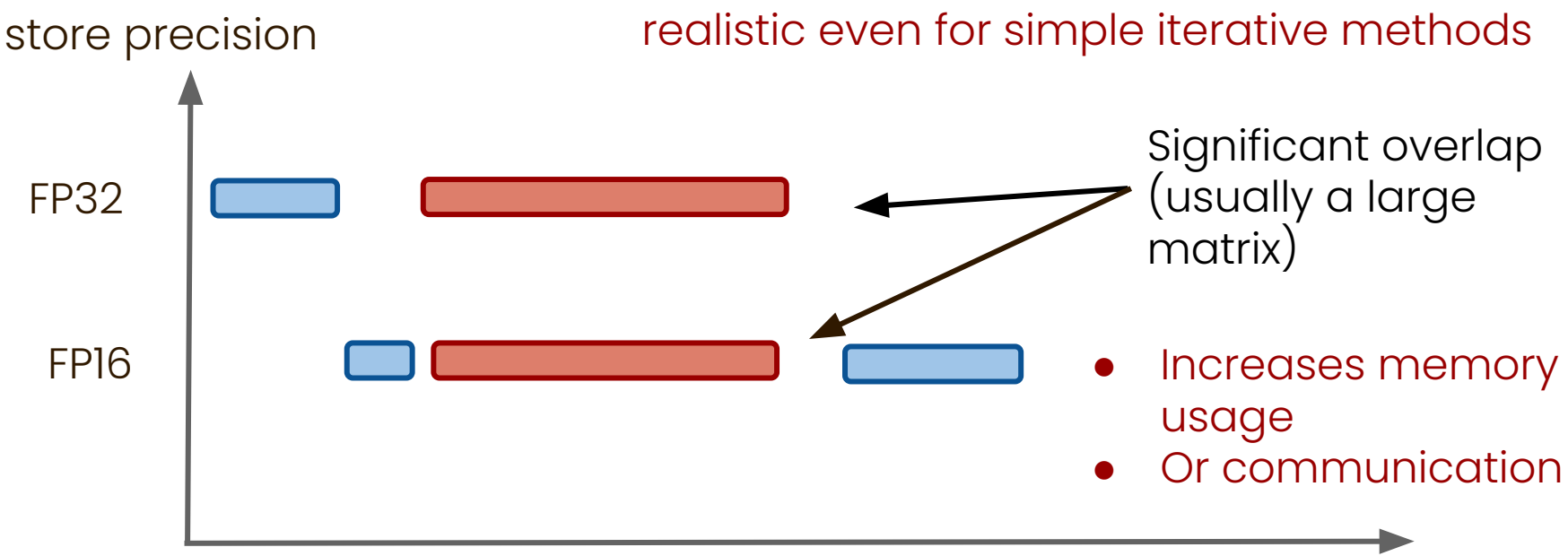
Memory usage of a numerical solver



Memory usage of a numerical solver



Memory usage of a numerical solver



RandNLA - a special case

Matrix sketching as a foundational tool

- powerful algorithmic technique

How to adapt for the HPC infrastructure?

1. Randomness generation tools
2. Sketch generation management, distribution
3. **Optimize the target matrix accesses**

Sketches can be generated online directly in local memory.
Bandwidth completely occupied by matrix element transfers.

Goal

A GPU-friendly format to encode contiguous blocks of memory:

1. Represent at least 2 floating point precisions (fp32 and fp16)

Goal

A GPU-friendly format to encode contiguous blocks of memory:

1. Represent at least 2 floating point precisions (fp32 and fp16)
2. General purpose - can reconstruct high-precision exactly

Goal

A GPU-friendly format to encode contiguous blocks of memory:

1. Represent at least 2 floating point precisions (fp32 and fp16)
2. General purpose - can reconstruct high-precision exactly
3. Low-precision retains its exponent range

Goal

A GPU-friendly format to encode contiguous blocks of memory:

1. Represent at least 2 floating point precisions (fp32 and fp16)
2. General purpose - can reconstruct high-precision exactly
3. Low-precision retains its exponent range
4. Memory usage = size(high-precision-obj)

Goal

A GPU-friendly format to encode contiguous blocks of memory:

1. Represent at least 2 floating point precisions (fp32 and fp16)
2. General purpose - can reconstruct high-precision exactly
3. Low-precision retains its exponent range
4. Memory usage = size(high-precision-obj)
5. Optimal number of memory transfers for every precision

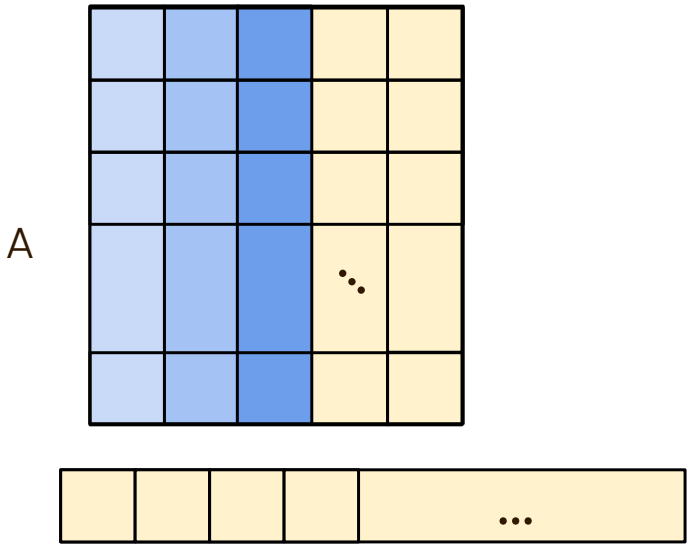
Goal

A GPU-friendly format to encode contiguous blocks of memory:

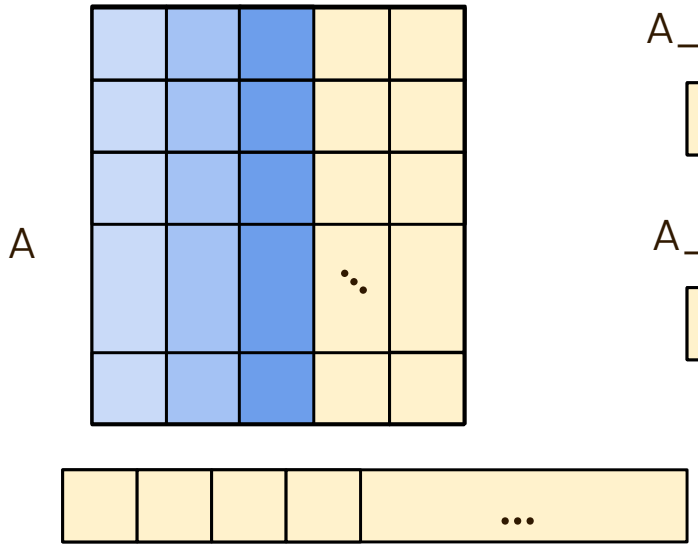
1. Represent at least 2 floating point precisions (fp32 and fp16)
2. General purpose - can reconstruct high-precision exactly
3. Low-precision retains its exponent range
4. Memory usage = size(high-precision-obj)
5. Optimal number of memory transfers for every precision

- Access through custom blas-kernels, potential overhead for high-precision operations -> overhead cost amortizes

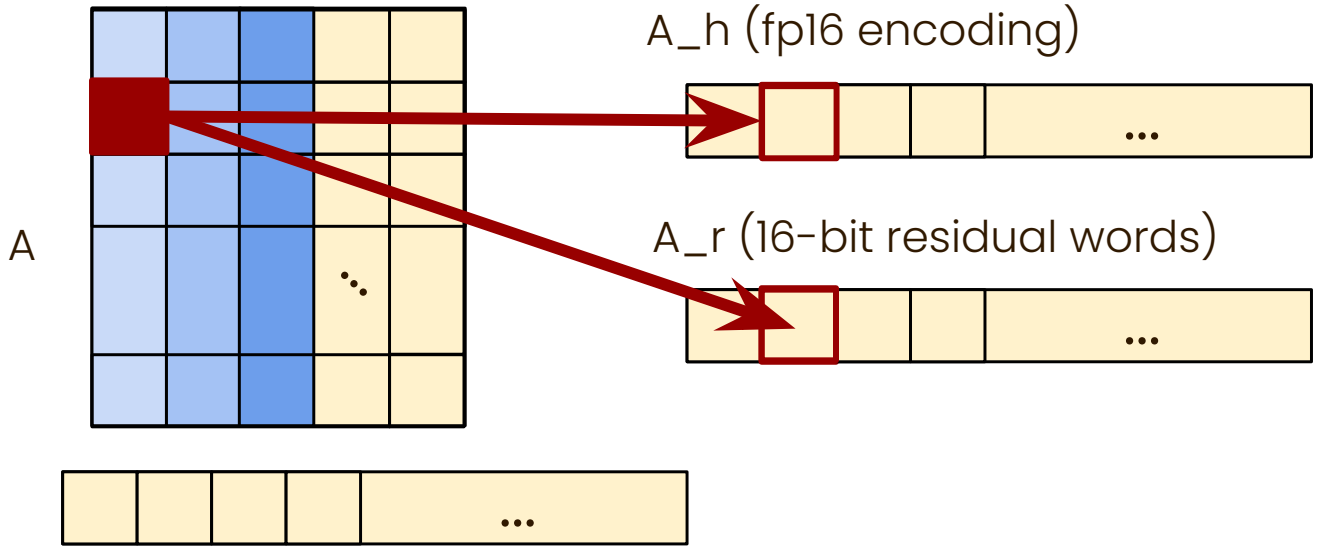
A composite format for multiprecision operations



A composite format for multiprecision operations



A composite format for multiprecision operations

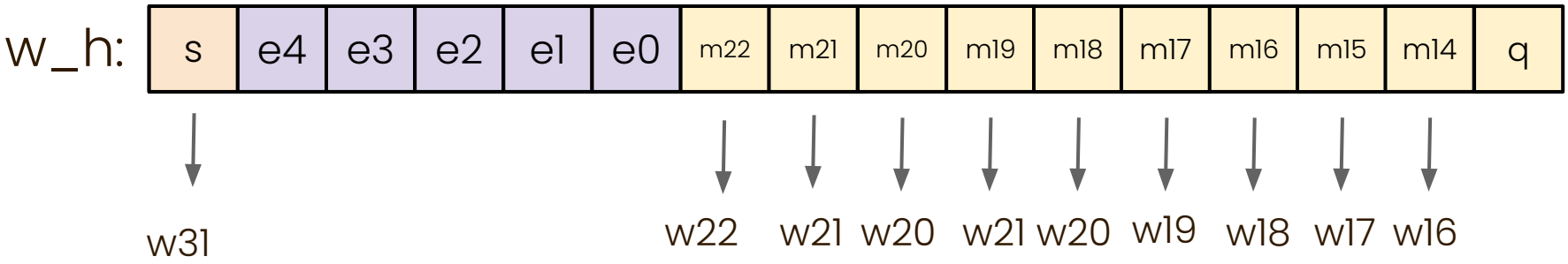


Encoding

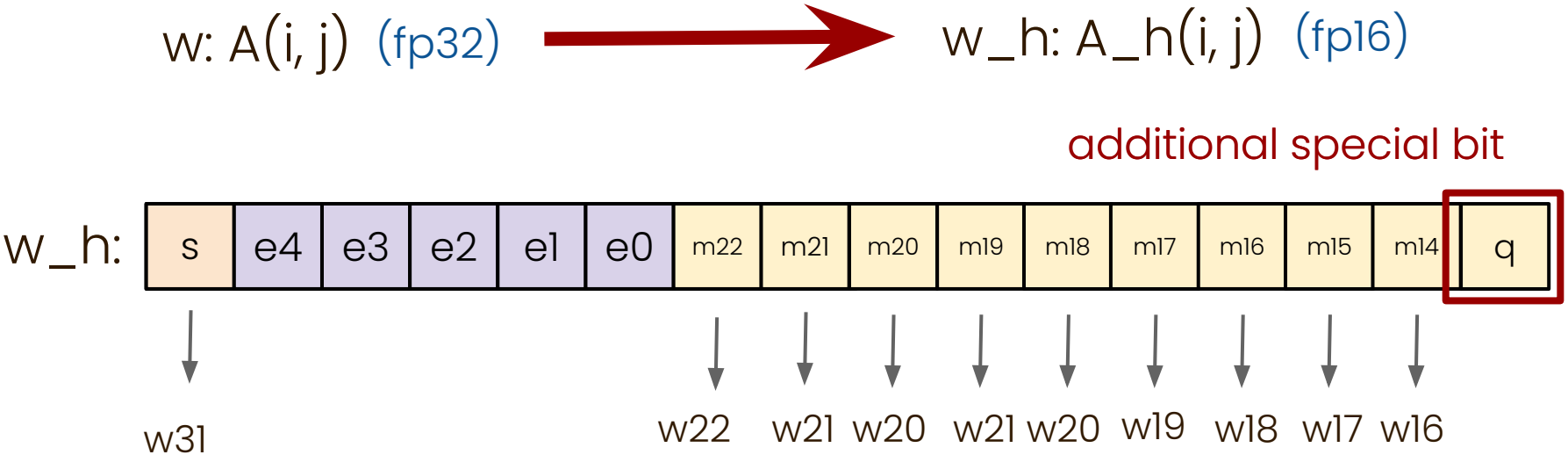


Encoding

w: A(i, j) (fp32) → w_h: A_h(i, j) (fp16)



Encoding

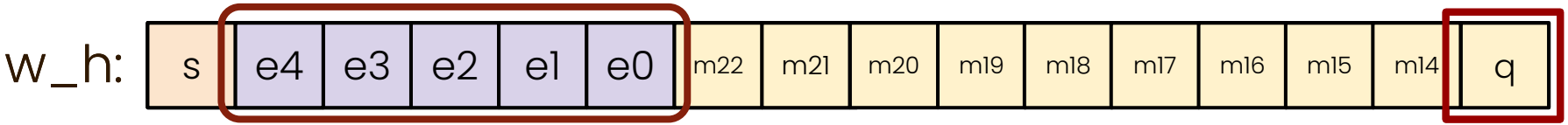

w: A(i, j) (fp32)
w_h: A_h(i, j) (fp16)
additional special bit
w_h:
s
e4
e3
e2
e1
e0
m22
m21
m20
m19
m18
m17
m16
m15
m14
q
w31
w22
w21
w20
w21
w20
w19
w18
w17
w16

Encoding

w: A(i, j) (fp32) → w_h: A_h(i, j) (fp16)

E

additional special bit



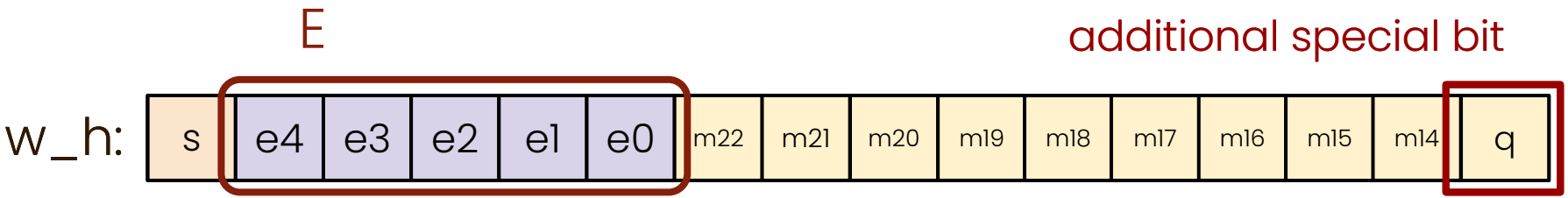
Controls the function of E

m0 = 0 → E = exponent of normal numbers, Inf, NaN

m0 = 1 → subnormals*

Encoding

w: A(i, j) (fp32) → w_h: A_h(i, j) (fp16)



Controls the function of E

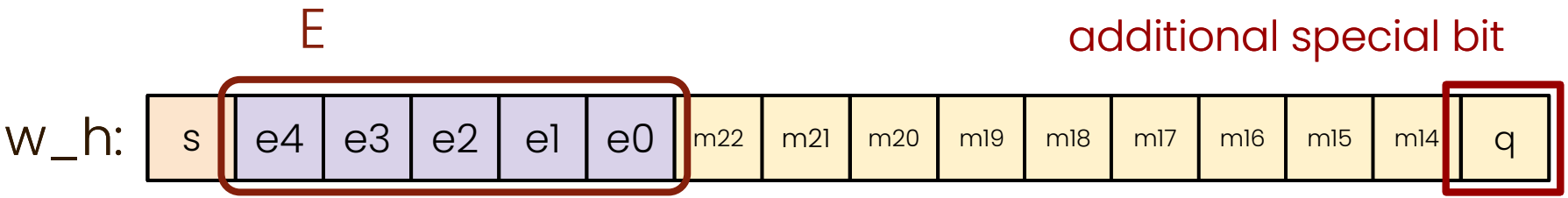
m0 = 0 → E = exponent of normal numbers, Inf, NaN

m0 = 1 → subnormals*

* if subnormal use E to encode the right shift!

Encoding

w: A(i, j) (fp32) → w_h: A_h(i, j) (fp16)



Controls the function of E

m0 = 0 → E = exponent of normal numbers, Inf, NaN

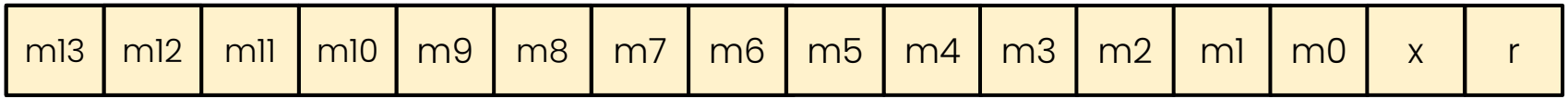
m0 = 1 → subnormals*

* if subnormal use E to encode the right shift!

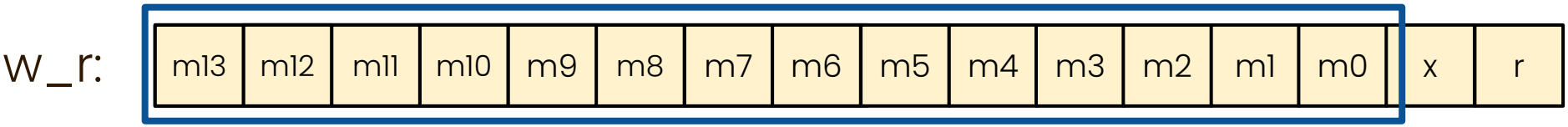
Can be encoded with m0 = 1 to reduce space

Residual word

w_r:

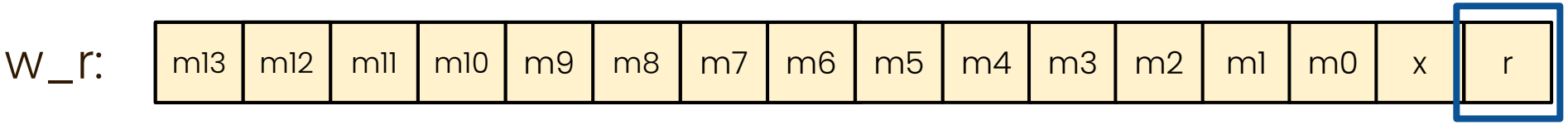


Residual word



14 mantissa bits (of the fp32 word)

Residual word



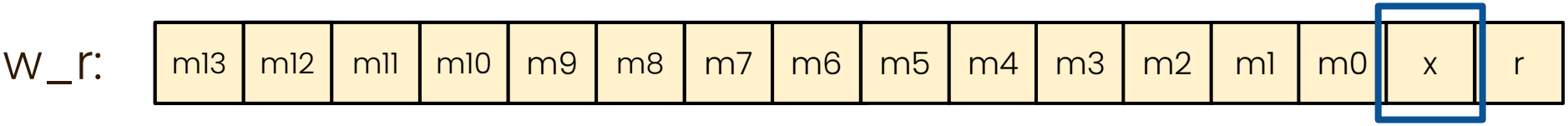
14 mantissa bits (of the fp32 word)

1 rounding bit

Mark round up/down during creation of w_h, to recreate the fp32 version



Residual word

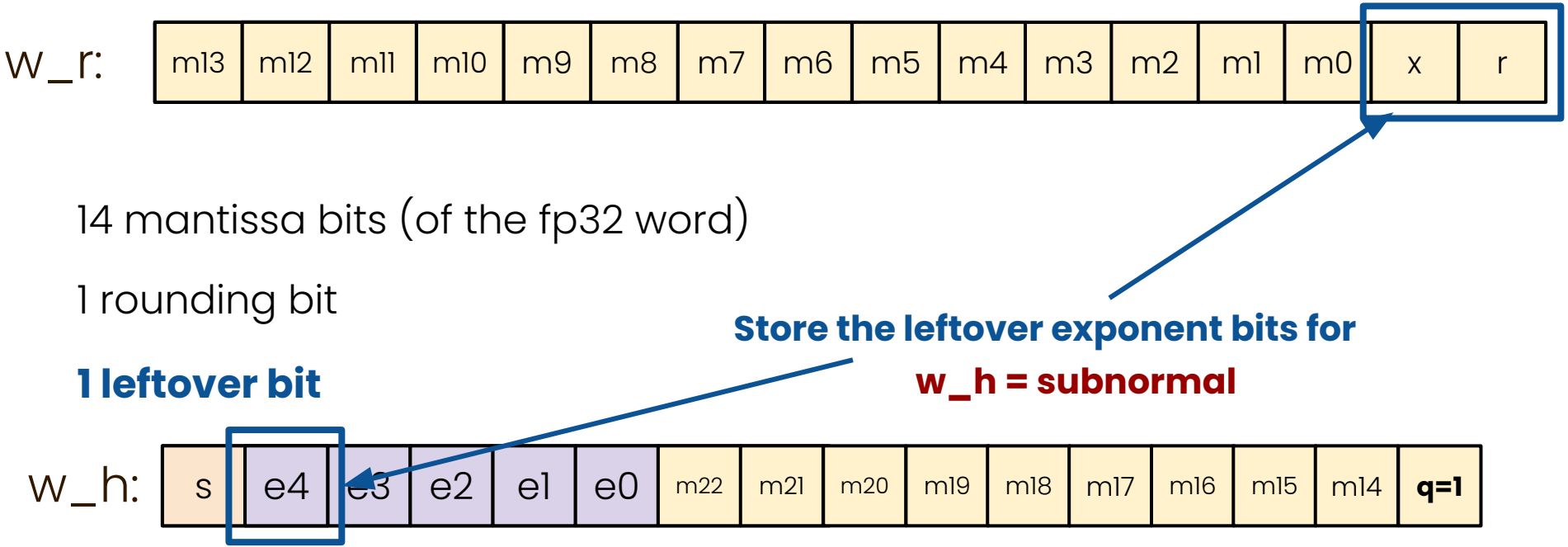


14 mantissa bits (of the fp32 word)

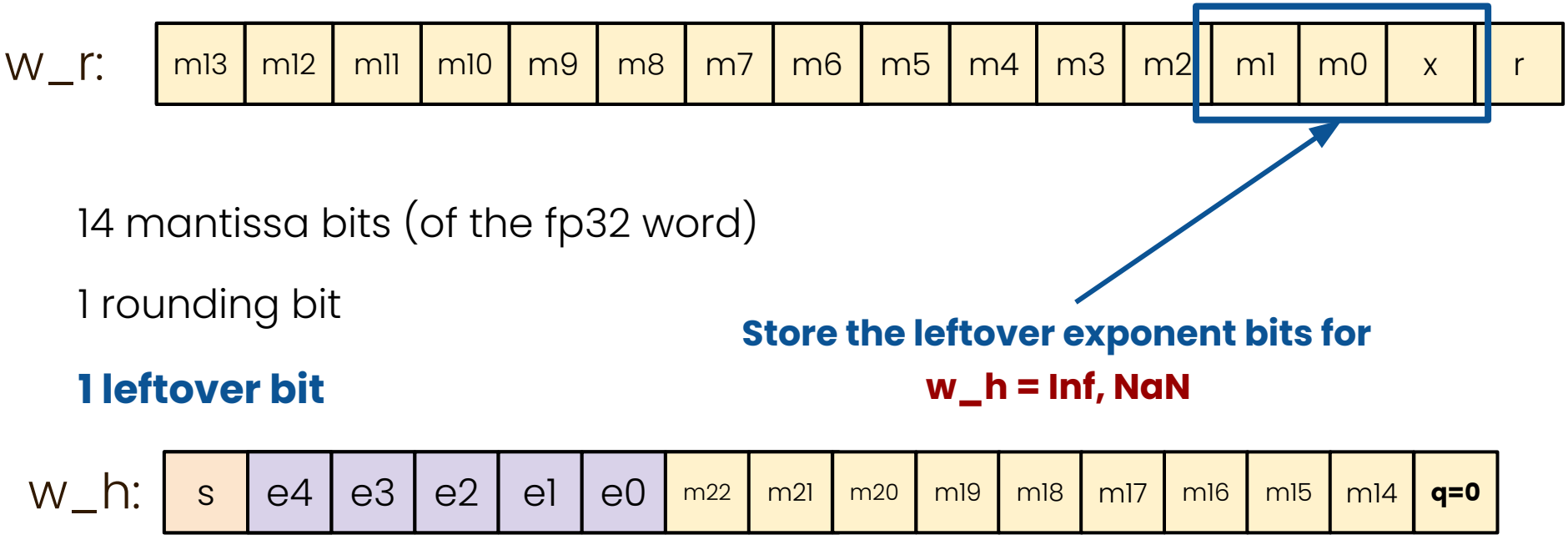
1 rounding bit

1 leftover bit Use it to recover from low-precision special values!

Residual word

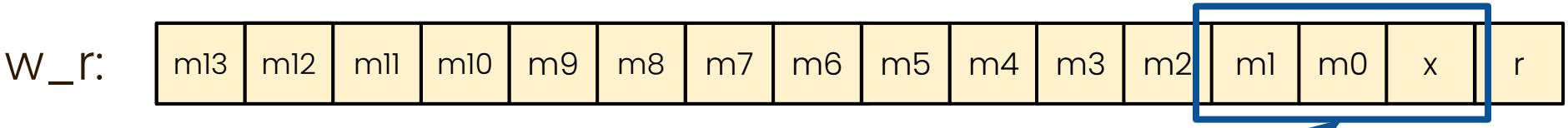


Residual word



Residual word

* This encoding will lose 2 bits of precision It can be avoided but leads to complicated decoding code



14 mantissa bits (of the fp32 word)

1 rounding bit

1 leftover bit

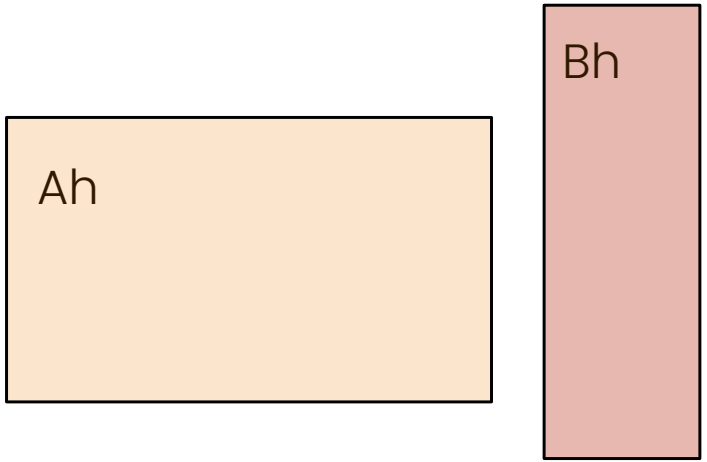
Store the leftover exponent bits for

w_h = Inf, NaN



Low-precision access through custom BLAS

Low-precision (fp16) contiguous values can be accessed by through w_h

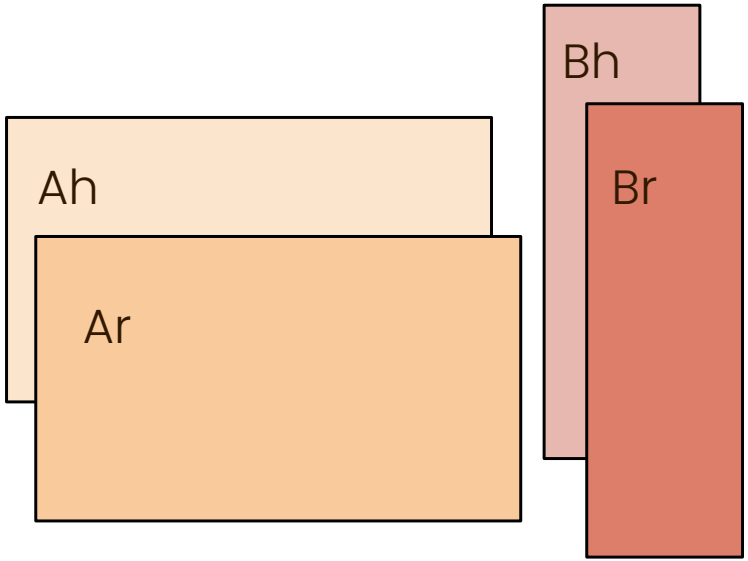


hgemm_w(Ah, Bh)

- internally calls pack_fp16(wh), only modifies subnormals
- close to zero overhead
- same throughput

High-precision access through custom BLAS

High-precision (fp32) contiguous values can be accessed by through w_h, w_r (for both A and B)



sgemm_w(Ah, Ar, Bh, Br)

- internally calls pack_fp32(wh, wr), only modifies subnormals
- more complicated
- same throughput but different number of load instructions => different optimization strategies

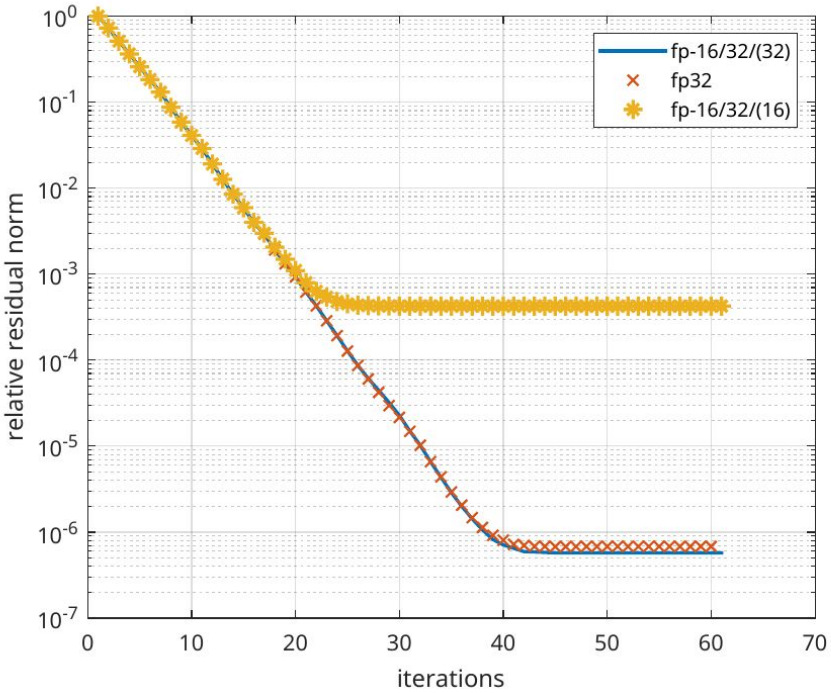
Some numerical examples

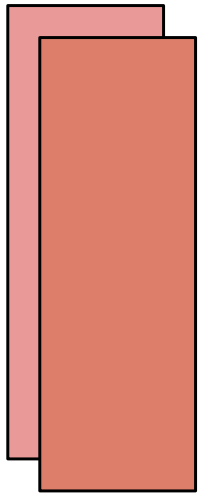
Focus on linear regression test problems sketch-and-precondition

Custom countsketch, cusolver used for qr factorization..
Custom gemv and lsqr implementation to solve the regression problem

- Composite format depicted as FP-16/32(X), X = 16, 32 the compute precision
- Synthetic test case.
- Easy to evaluate performance, accuracy of mixed precision code

Numerical example: A = 1e-4 * randn(1e5, 4e2)



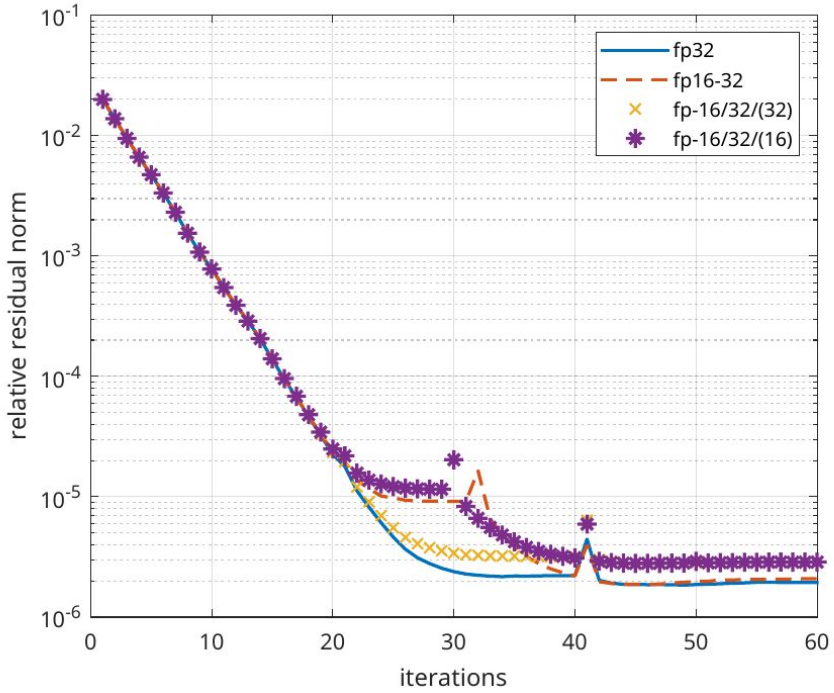


Stored in
FP32 or FP16

FP-16/32/(32) - 32-bit MVs
FP-16/32/(16) - 16-bit MVs
FP-16-32 - 16-bit MVs

FP-16/32/(32) - achieves FP32 Precision - exact decode

Numerical example: A = 1e-4 * randn(1e5, 4e2)



20 iterations, 3 restarts, 50 runs

precision	runtime	residual	Memory usage
FP-32	1.2E-1	**1.9E-6**	**1.2MB**
FP-16-32	**8.9E-2**	**1.9E-6**	160.1MB
FP-16/32/(32)	1.3E-1	2.7E-6	**1.2MB**
FP-16/32/(16)	**8.9E-2**	3.4E-6	**1.2MB**

~15% overhead for encode/decode

~30% speedup, 98.5% workspace memory reduction, 33% mem reduction

Conclusions

- Use a GPU-friendly matrix format to efficiently encode multiple precisions (here fp32 and fp16)
- Allow for optimal amount of memory usage and communication
- Added overhead in encoding/decoding steps of
- Optimizations: dropping subnormals? Truncation vs rounding?

Thank you

Questions?