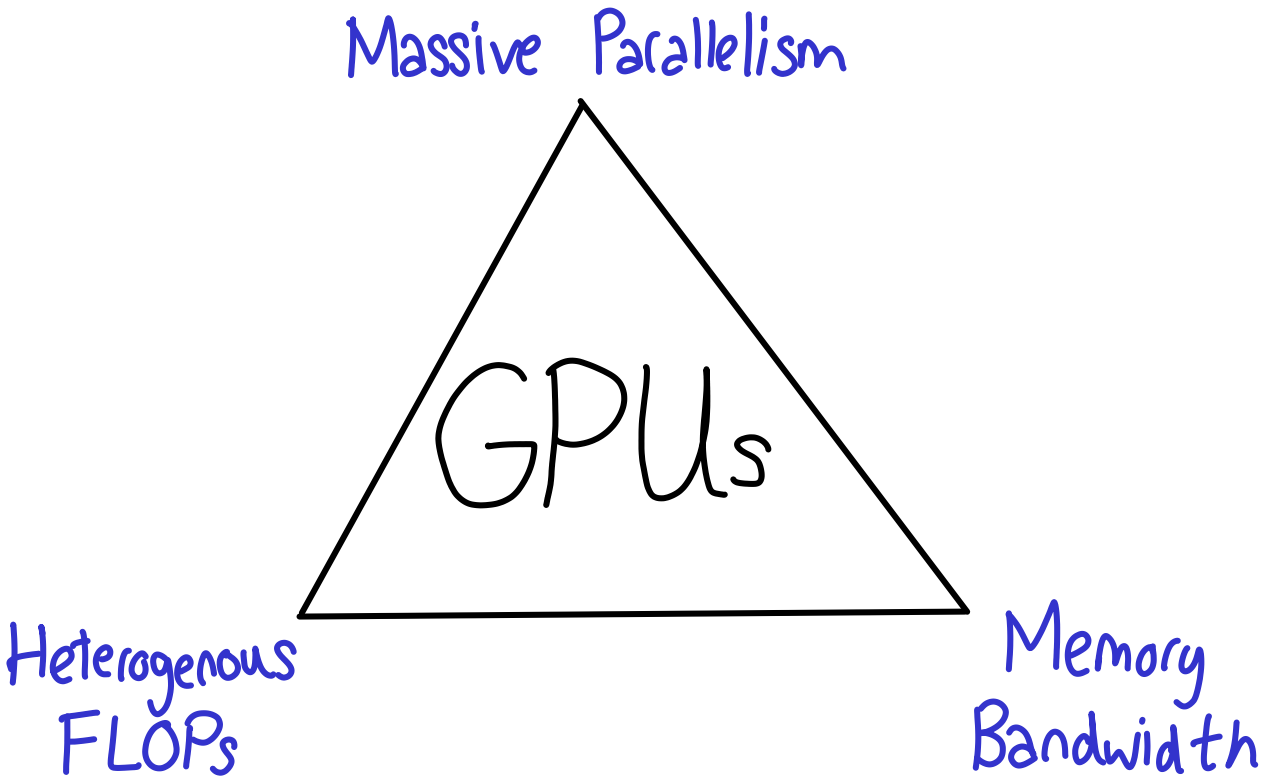
GPUs RULE EVERYTHING AROUND ME

Edward Z. Yang

Deep learning...

... a strange parallel universe to scientific computing

Why so strange?



Example: NVIDIA's upcoming Hopper architecture

Technical Specifications	
	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

big! 3x uniform increase

NB: different types of FLOPs!

not as big as you might think! only 1.5-2x increase

Why so strange?

(Another answer...)

rarely. but NNs are quite good at adapting to low prec via training

e.g. Courbariaux 2015 (BinaryConnect)
Narang 2018 (Mixed Precision Training)

Massive Parallelism

Example: H100 non-Tensor core FP32 TFLOPs

2 FP32 ops/cycle
× 128 FP32 cores per SM
× 132 SMs
× 1980 MHz

16896 cores/GPU

66.9 peak FP32 TFLOPs

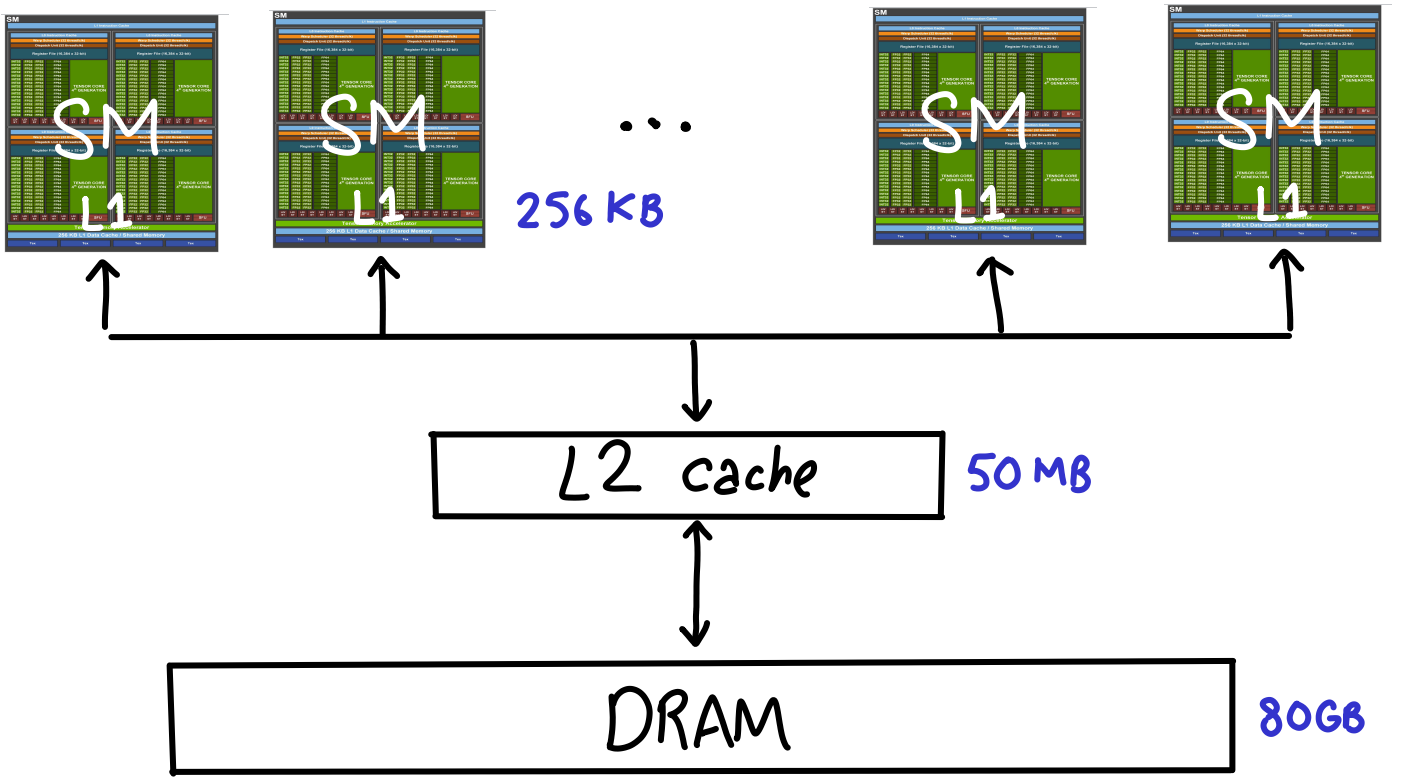
↑ not so easy to actually achieve!

Massive Parallelism

all threads execute in lockstep: data dependent control flow bad for perf!

shared memory PAIN !!!

Massive Parallelism



Plan algorithms accordingly...

e.g. cascade summation #39516

Heterogenous FLOPs

More to life than H100s ($40k a pop!)

consumer cards like GeForce 16xx/20xx

not pictured: yes int32 cores, no int64 cores

not FP32!

	CUDA Cores				Tensor Cores					
NVIDIA Architecture	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64 *	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

this is really small!

Tensor Cores!

- reluctance to use rlibm
- accumulation in fp32, not fp64

* better to do FP64 matmul than INT64 matmul! (CrypTen)

Heterogenous FLOPs: Tensor Cores

Motivation: Matrix multiplies are arithmetically expensive
General purpose FLOPs harder than special silicon...

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

$D = A \times B + C$

where **A**, **B**, **C**, and **D** are 4x4 matrices (Figure 8). The matrix multiply inputs **A** and **B** are FP16 matrices, while the accumulation matrices **C** and **D** may be FP16 or FP32 matrices (see Figure 8).

Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

always low precision, even w/ FP32 inputs!
10 bit mantissa only; accumulate in FP32

Heterogenous FLOPs: Tensor Cores

spoiler alert: no!

RFC: Should matmuls use tf32 by default? #67384

TensorFloat32 tensor cores

Closed ngimel opened this issue on Oct 27, 2021 · 15 comments

ngimel commented on Oct 27, 2021 • edited by pytorch-probot bot Contributor

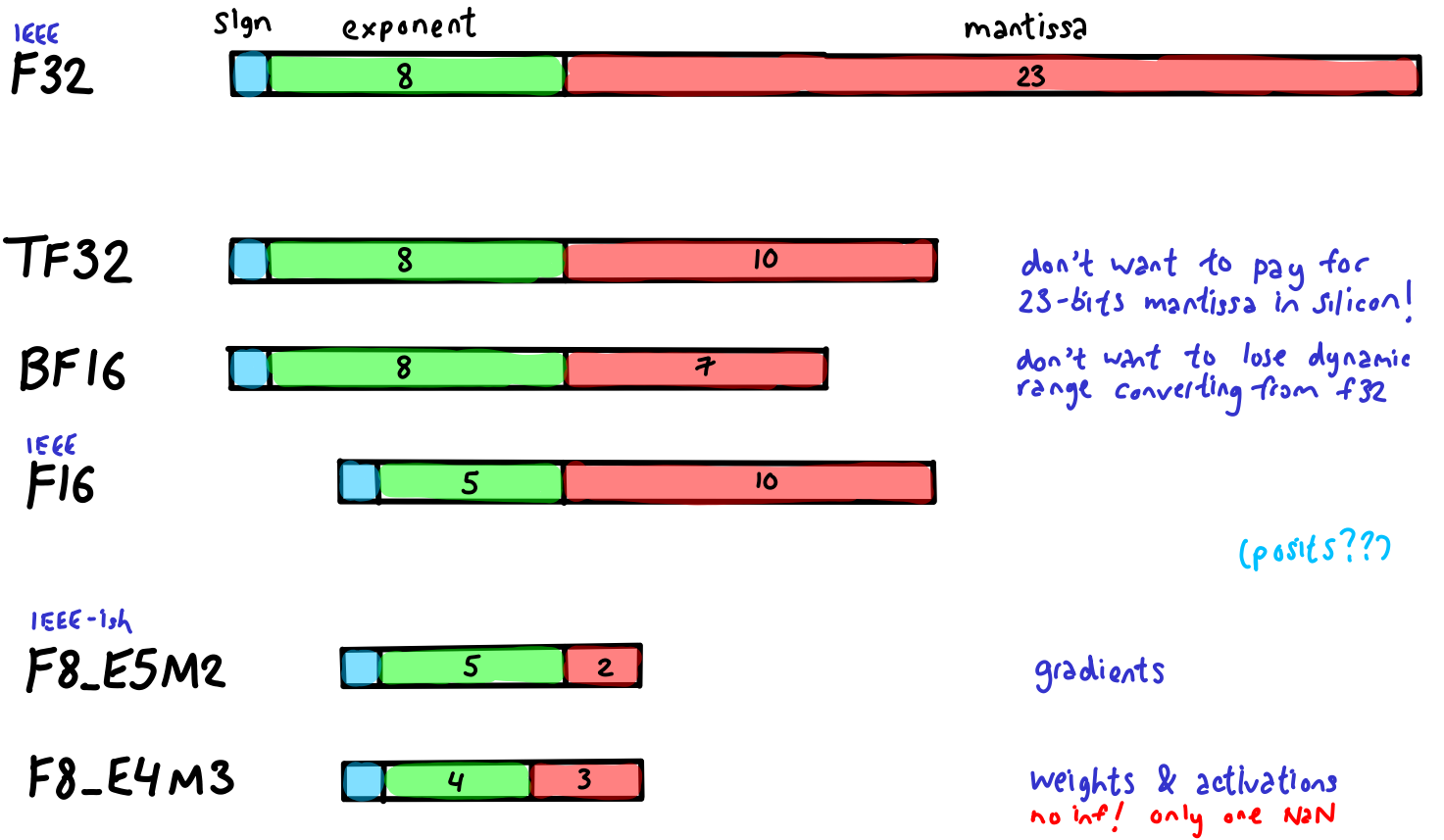
Since the release of Ampere GPUs, pytorch has been using tf32 by default. It is providing much better performance at the expense of somewhat lower accuracy. Nvidia has conducted a lot of experiments proving that convergence behavior of a wide variety of networks does not change when tf32 is used instead of regular fp32.
However, since pytorch is used not only for deep learning workloads, or for some non-standard deep learning workloads, the use of tf32 for matrix multiplication has resulted in a lot of confusion and sometimes bad results.

Bug? matmul seems to cast to float16* internally

* not actually float16!

The matrix multiplication operator can't get correct results on 3090 !!

Heterogenous FLOPs: Low precision



Heterogenous FLOPs: NVIDIA F8_E4M3

mantissa →

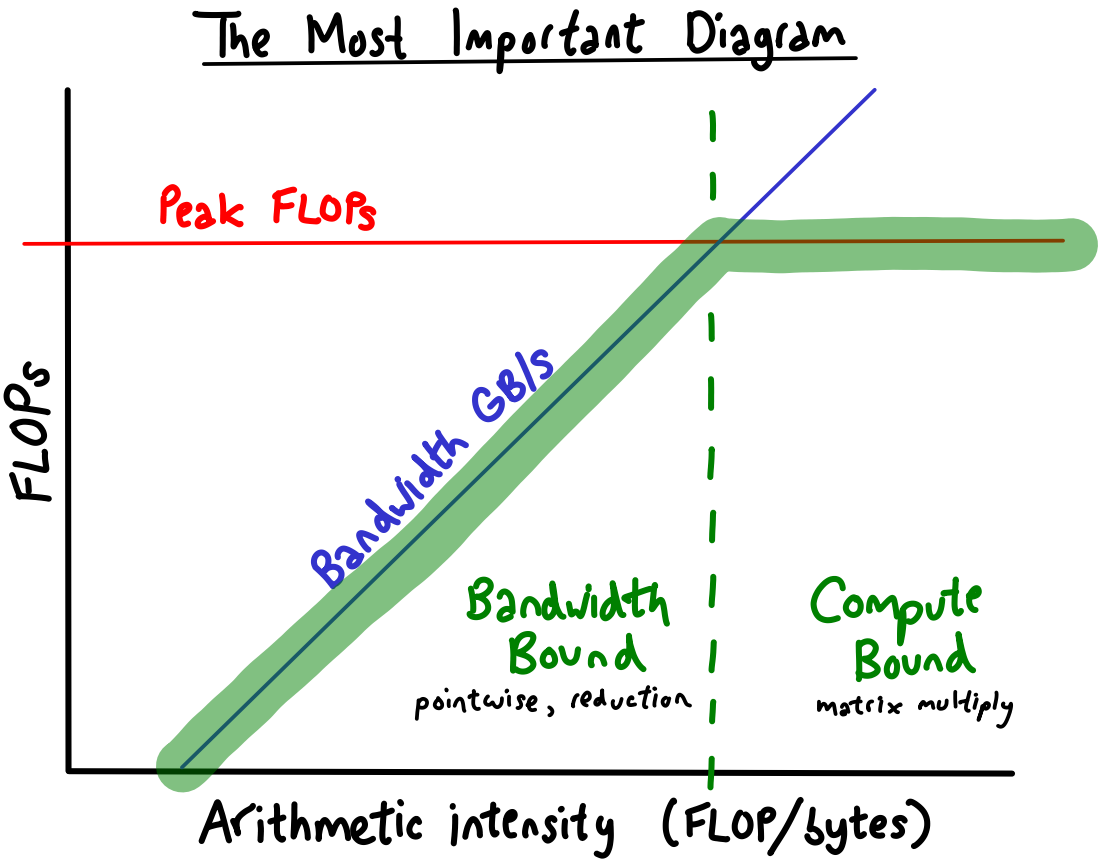
Subnormals

0	0.00195312	0.00390625	0.00585938	0.0078125	0.00976562	0.0117188	0.0136719
0.015625	0.0175781	0.0195312	0.0214844	0.0234375	0.0253906	0.0273438	0.0292969
0.03125	0.0351562	0.0390625	0.0429688	0.046875	0.0507812	0.0546875	0.0585938
0.0625	0.0703125	0.078125	0.0859375	0.09375	0.101562	0.109375	0.117188
0.125	0.140625	0.15625	0.171875	0.1875	0.203125	0.21875	0.234375
0.25	0.28125	0.3125	0.34375	0.375	0.40625	0.4375	0.46875
0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
1	1.125	1.25	1.375	1.5	1.625	1.75	1.875
2	2.25	2.5	2.75	3	3.25	3.5	3.75
4	4.5	5	5.5	6	6.5	7	7.5
8	9	10	11	12	13	14	15
16	18	20	22	24	26	28	30
32	36	40	44	48	52	56	60
64	72	80	88	96	104	112	120
128	144	160	176	192	208	224	240
256	288	320	352	384	416	448	nan

↑ no infinity, only one NaN

exponent ↓

Memory Bandwidth



Memory Bandwidth

If you are BW-bound, more compute is free!

Recall the cores:

- INT32
- FP32
- FP64

} no dedicated non-TensorCore for FP16/BF16!

↳ primarily about bandwidth savings (internal compute done in FP32)

↳ fusing FP16 ops increases overall prec (truncating would harm acc & perf!)

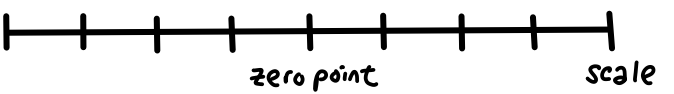
- TensorCore

Matrix multiply compute bound: every dtype has its own core type

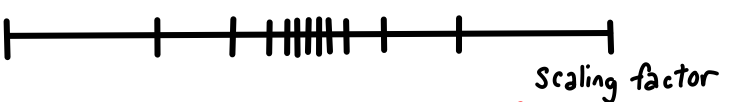
Memory Bandwidth: Quantization

lots of complexity:
dynamic or static range?
per-tensor / channel / ... ?

INT8 quantization

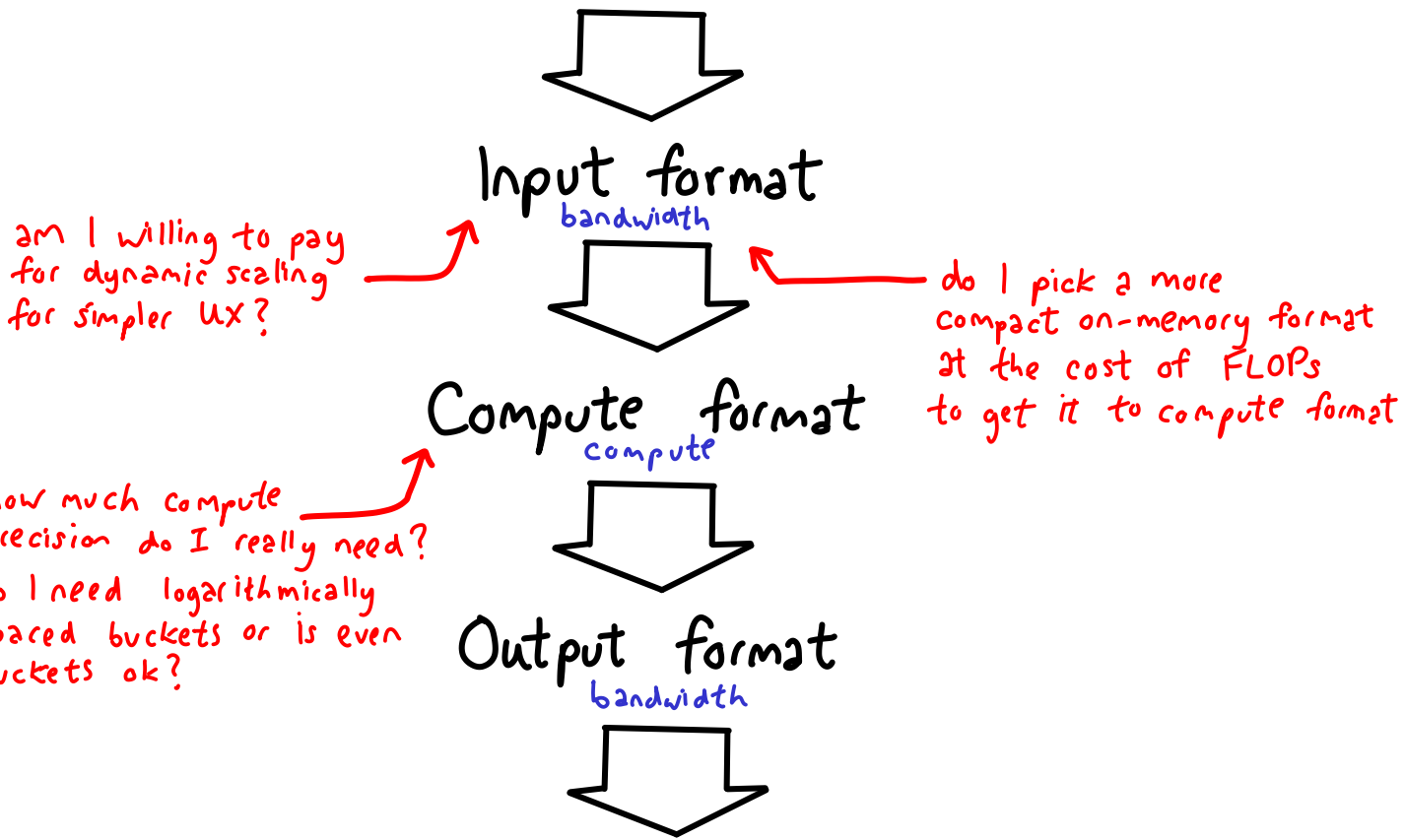


FP8



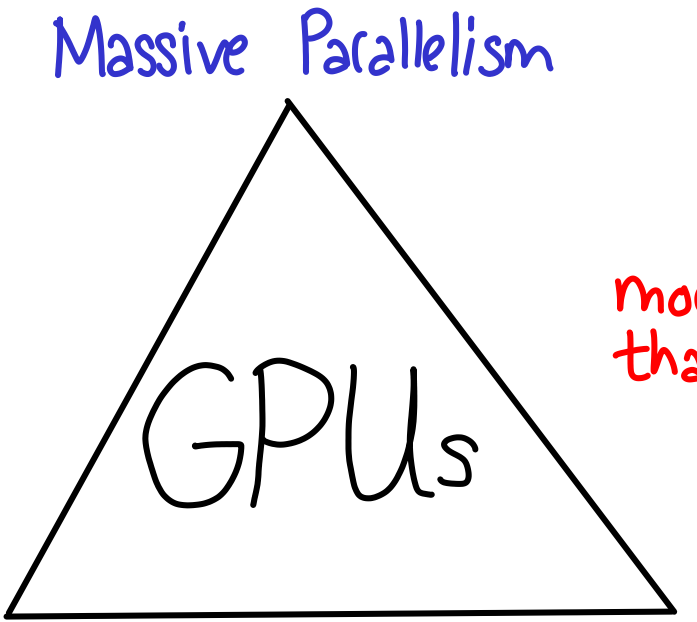
research problem!

Memory vs. Compute tradeoff



Conclusion

data size makes tooling challenging



more to life than accuracy!

quantization / super low precision/sparsity

Bonus slides

- Ordinary FP snafus
  - Grid sampling #24832
  - Interpolate scale factor #93598
  - Softmax doesn't sum to 1 #101039
- Normal FP lore
  - Epsilons
  - NaN correctness
  - FMA
- Compiler correctness: FP64 reference

Bonus slides

- Vanishing gradients in NNs
- Posits?
- Sympy floating point bugs (related to interpolate)