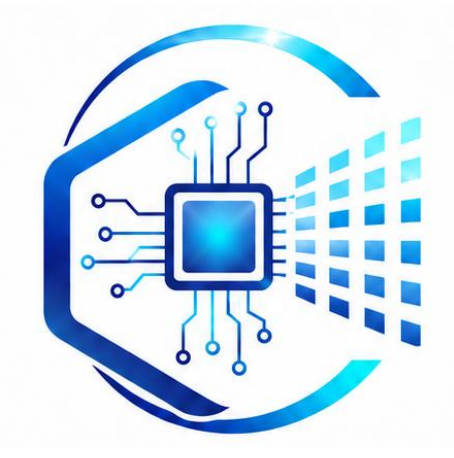


【豊島高等学校・模擬講義】

AIを^{かしこ}賢くするための 高速計算技術

2026年10月15日（木）



追手門学院大学
理工学部 数理・データサイエンス学科
幸谷(こうや) 智紀(とものり)

<https://kd-tklab.na-inet.jp/>



講義概要

- 「賢い」って何だろう？
- AI（人工知能）の動作原理
- 計算を高速に実行する仕組み(1/2)：計算を軽くする
 - 計算する回数を減らす
 - 計算桁数を短くする
- 計算を高速に実行する仕組み(2/2)：並列に処理する
 - マルチコアCPU
 - メニーコアGPU→NPU, TPU
- AIによる分類問題のデモ
- まとめ

「賢(かしこ)い」って何だろう？

- 「あの人は頭が良い」とは？
 - どんな疑問に対しても的確な回答ができる = 正確さ
 - 回答に要する時間が短い = 速さ
- AI = 人工(Artificial)知能(Intelligence) も同じ
 - どんな疑問に対しても的確な回答ができる
 - LLM = 大規模(Large)言語(Language)モデル(Model)：大量の文章から学習
 - 回答に要する時間が短い
 - スーパーコンピュータ(Super Computer)を使って高速処理
- この講義のテーマ：「速さ」はどうやって手に入れるのか

LLMの例

アメリカ産

- OpenAI ChatGPT(チャッピー) / Anthropic Claude(クロード)
- Google Gemini(ジェミニ) / Microsoft Copilot(コパイロット) / X Grok(グロック)

• 中国産

- DeepSeek(ディープシーク) / Kimi(キミ) / Qwen(クウェン)

• 日本産

- PFN PLaMo / ELYZA / Sakana AI ほか

• 見た目も得意分野も違うが、中身の計算はどれも同じ

- = 巨大な行列（数の表）の掛け算を、ひたすら繰り返しているだけ

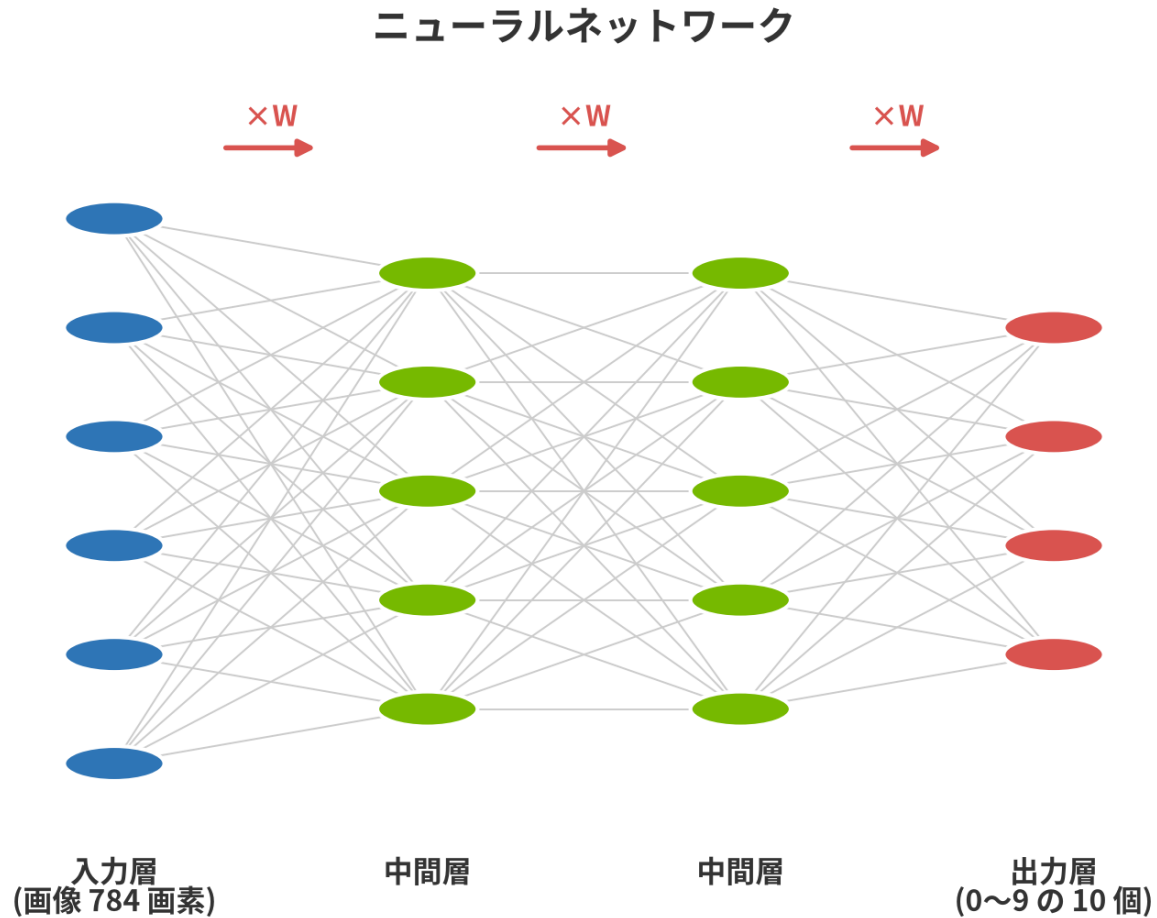
どれくらいの計算をしているのか？

- LLM の「パラメータ」 = 行列の中に並んだ数の個数
 - 大きなモデルでは 1000 億個 (10^{11}) 以上ある
- 文章を 1 語 (1 トークン) 作るのに必要な計算回数
 - **およそ $2 \times \text{パラメータ数} = 2 \times 10^{11}$ 回 = 2000 億回**
- この 1 語分を計算するのにかかる時間 (本日の実測性能で単純計算)
 - CPU (binary32, 0.84 TFLOPS) … 約 239 ミリ秒 / 1 語
 - **GPU (bfloat16, 99.2 TFLOPS) … 約 2.02 ミリ秒 / 1 語**
- **400 字の文章なら, CPU で約 96 秒 / GPU なら約 0.8 秒**
- 「待たされない AI」を作るには, 計算を速くするしかない

AI（人工知能）の動作原理

- 機械学習 (Machine Learning)
 - 人がルールを書くのではなく、大量のデータから規則を自動で見つけさせる
 - 例：手書き数字の画像 6 万枚を見せて「これは 3」と当てられるようにする
- 深層学習 (Deep Learning)
 - 脳の神経細胞をまねた「ニューラルネットワーク」を何層も重ねたもの
 - 層を深くするほど複雑な特徴をとらえられる（顔・言葉・音…）
- 2つの段階
 - 学習 (training)：正解を見ながら、行列の中の数（重み）を少しずつ直す
→ 重い
 - 推論 (inference)：できあがった行列を使って答えを出す → 軽いが回数が
多い
- どちらの段階も、計算の中身は「行列 × ベクトル」

ニューラルネットワークの正体は「行列 × ベクトル」



計算の中身

$$\begin{array}{c} W \\ \text{重み行列} \end{array} \times \begin{array}{c} x \\ \text{入力} \end{array} = \begin{array}{c} y \\ \text{出力} \end{array}$$

AI の計算 = 行列 × ベクトルの繰り返し

だから「行列積を速くする」ことが
そのまま「AI を速くする」ことになる

○ が「ニューロン」、線が「つながりの強さ（重み）」。重みを全部並べた数の表が 行列 W 。
1 つの層の計算は $y = \text{ReLU}(Wx + b)$ 。層を重ねる = 行列の掛け算を繰り返す、ということ。

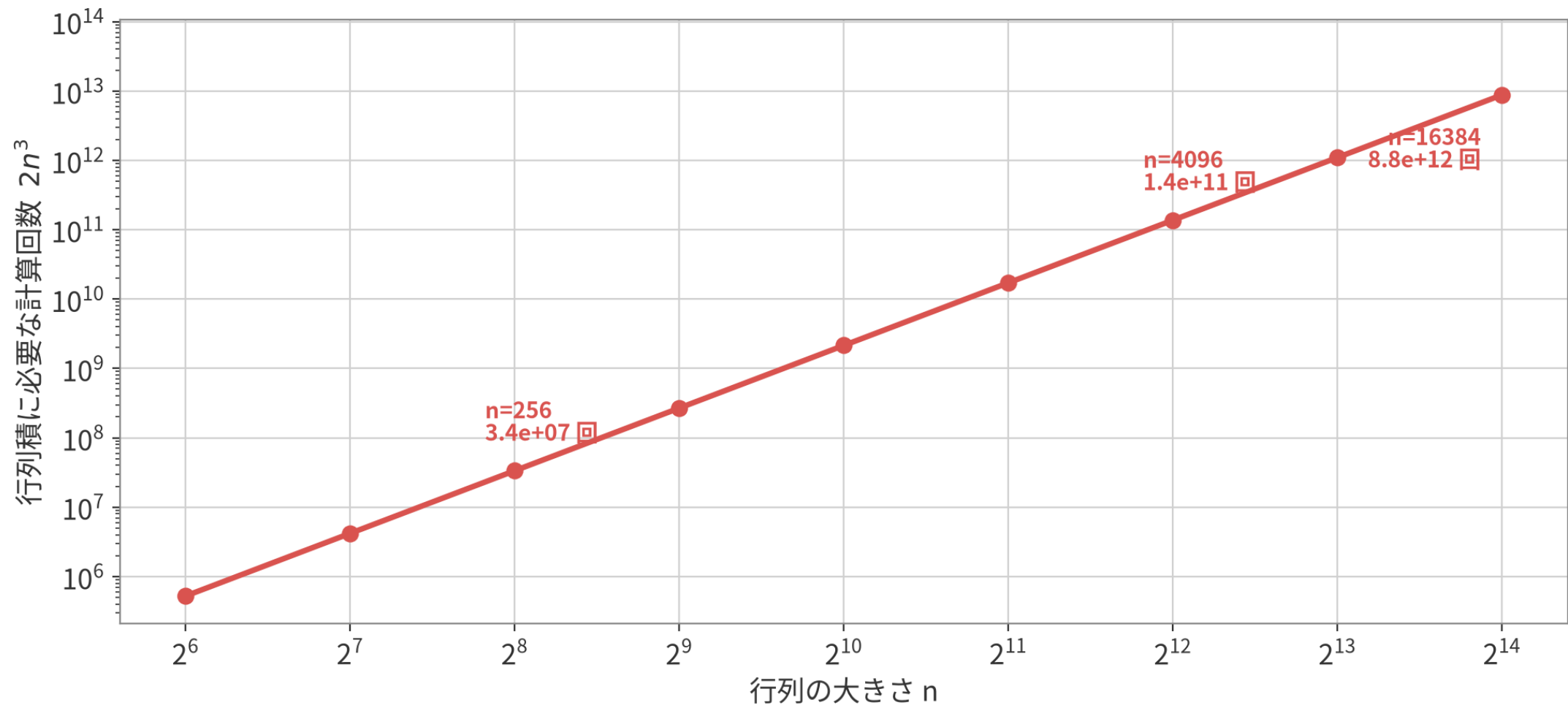
計算を高速に実行する仕組み(1/2)

計算を軽くする

- 考え方：同じ答えを出すのに，そもそも仕事量を減らせないか？
- **計算する回数を減らす**
 - 計算の順序を変える（数学的に等しい別の手順を選ぶ）
 - うまいアルゴリズムを使う（Strassen の行列積 など）
- **計算桁数を短くする**
 - 1 回の計算を安く・小さくする（64 bit → 16 bit）
 - 「必要な精度はどこまでか」を見極めるのが数値解析の仕事
- **AI は，実は「そこそこ正確」で十分な計算だった ← ここが効く**

計算する回数を減らす① —— まず、どれだけ大変かを知る

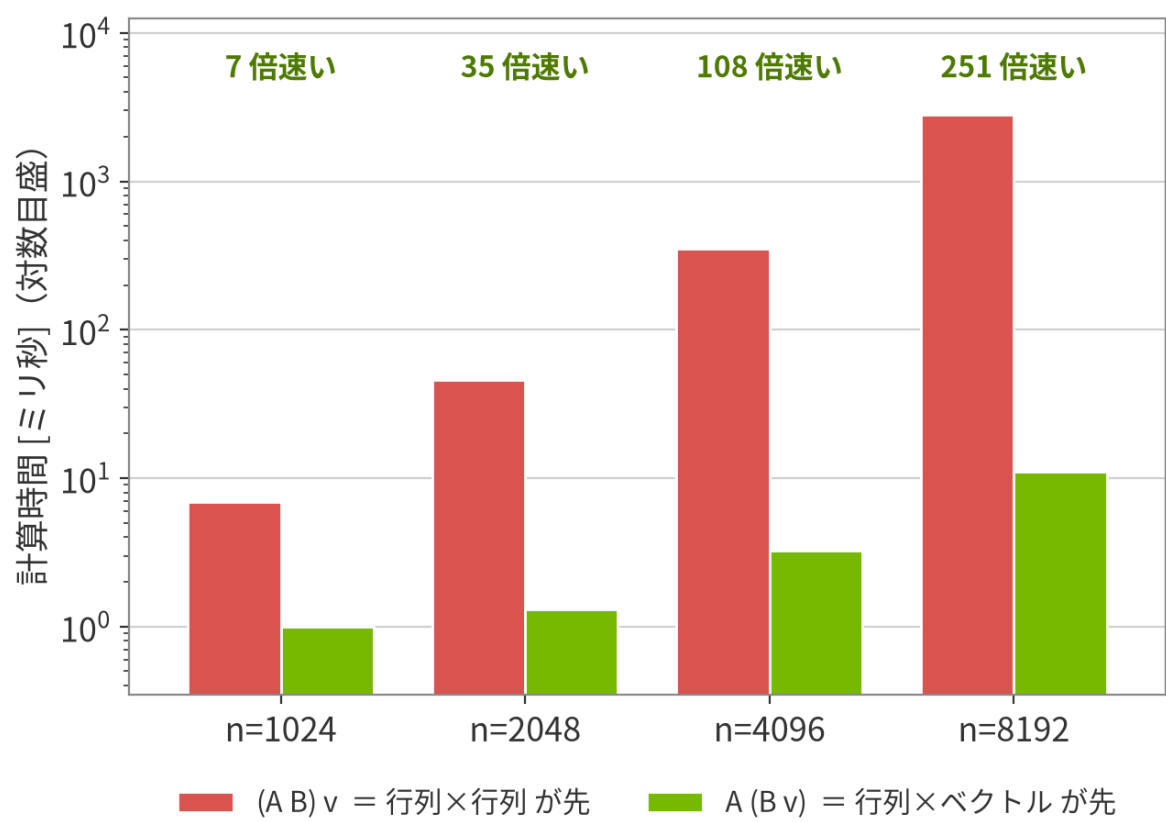
n が 2 倍になると計算回数は 8 倍に —— だから工夫が要る



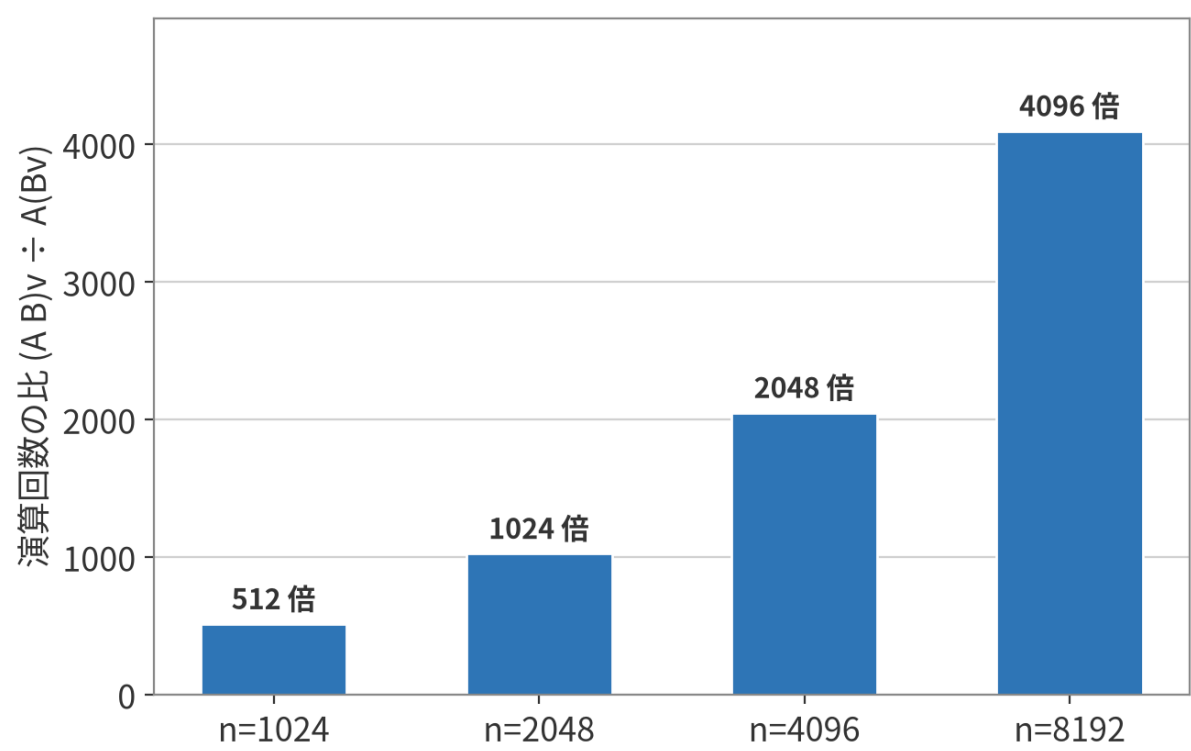
$n \times n$ の行列の掛け算に必要な計算回数は $2n^3$ 回。 n が 2 倍になると計算量は 8 倍。
 $n = 8192$ なら約 1.1 兆回。人が 1 秒 1 回計算しても 3 万 5 千年かかる。

計算する回数を減らす② —— 順序を変えるだけで速くなる

同じ答えなのに, かかる時間が違う



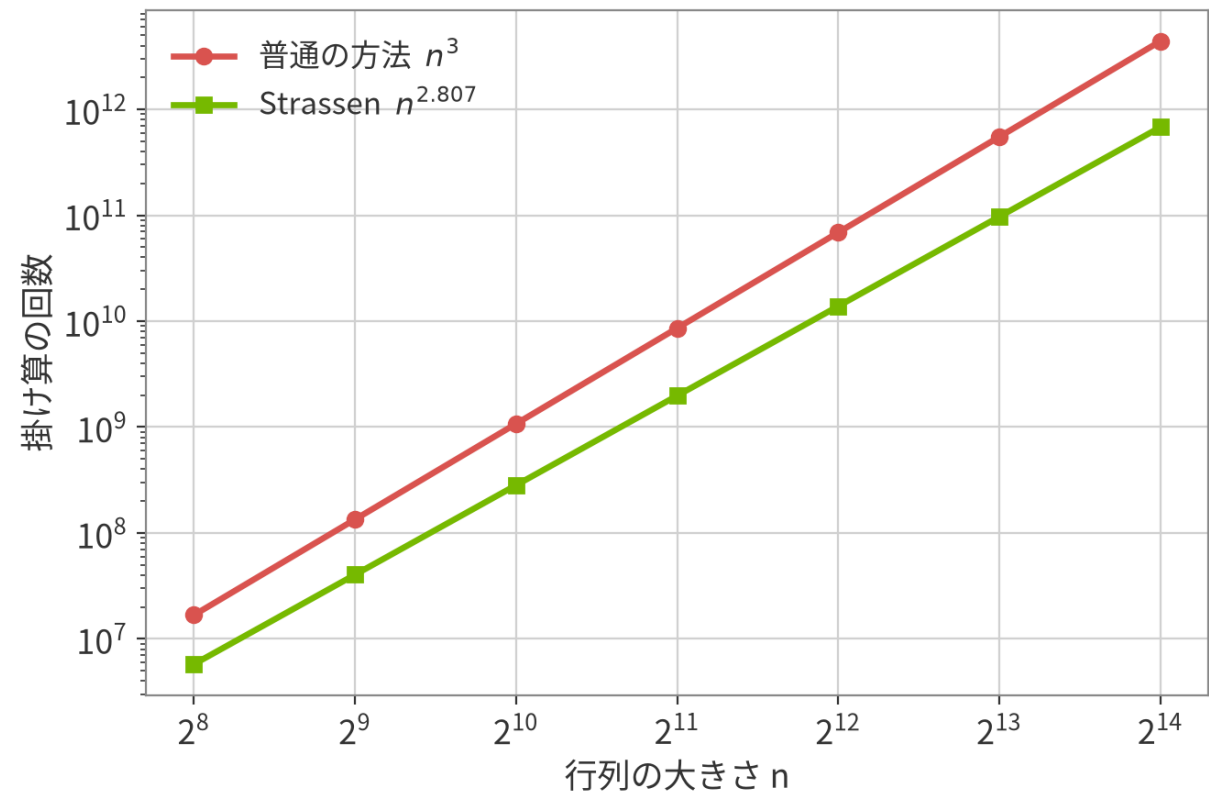
計算回数そのものが n/2 倍も違う



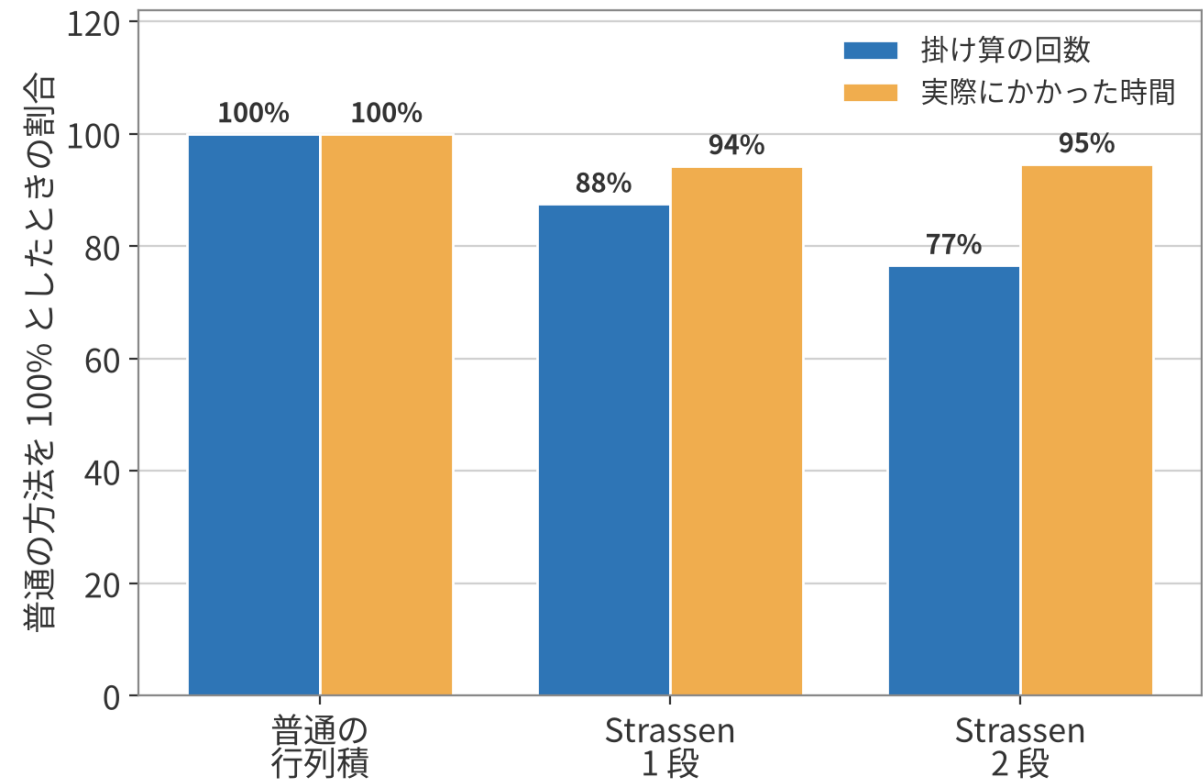
$(AB)v$ と $A(Bv)$ は数学的に同じ答え。しかし前者は $2n^3$ 回, 後者は $4n^2$ 回。
実測 (CPU, binary64, $n=8192$) : 2788.7 ミリ秒 → 11.1 ミリ秒 (251 倍速, 答えの差は $5e-15$ = 誤差の範囲)

計算する回数を減らす③ —— 賢いアルゴリズム，でも万能ではない

理論上の掛け算の回数



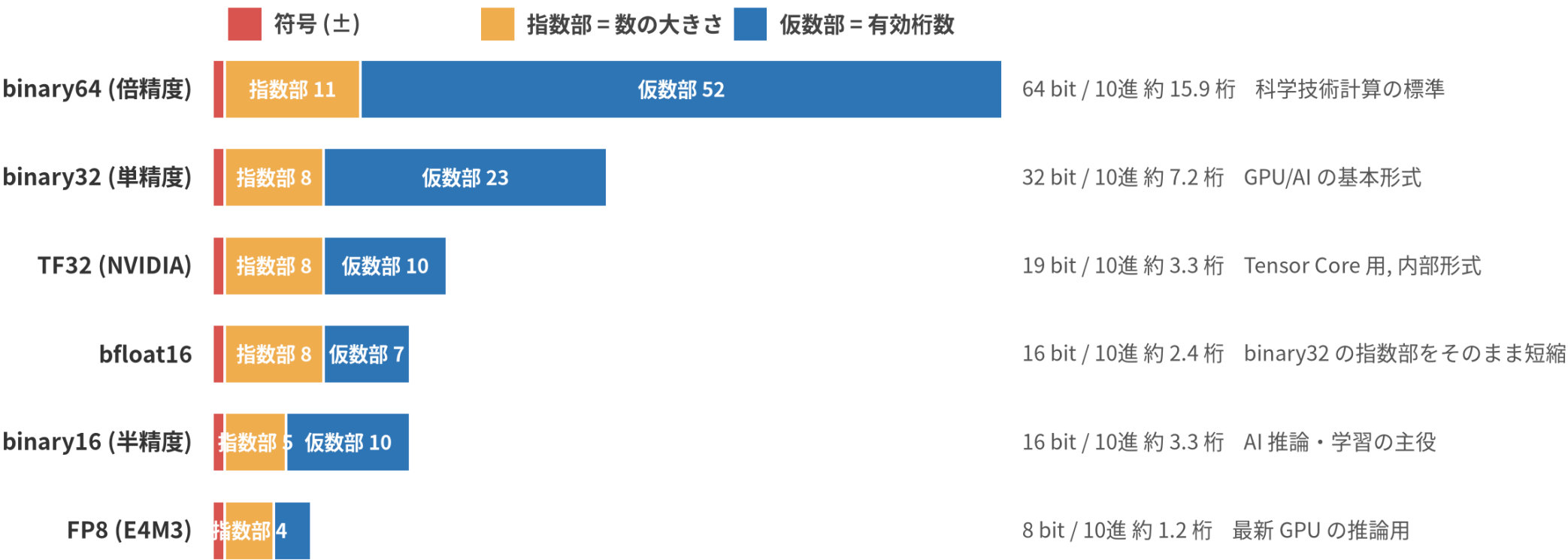
実測 (CPU, binary64, n=8192)
回数は減っても速くなるとは限らない



Strassen の方法：2×2 の行列の積を，掛け算 8 回ではなく 7 回で計算する（足し算は増える）。
再帰的に使うと，計算回数は n^3 から $n^{2.807}$ に減る。
ところが実測（ $n=8192$ ）では，掛け算を 23% 減らしても速さは 1.06 倍どまり。足し算とデータ移動が増えるため。

計算桁数を短くする①—— コンピュータの数の表し方

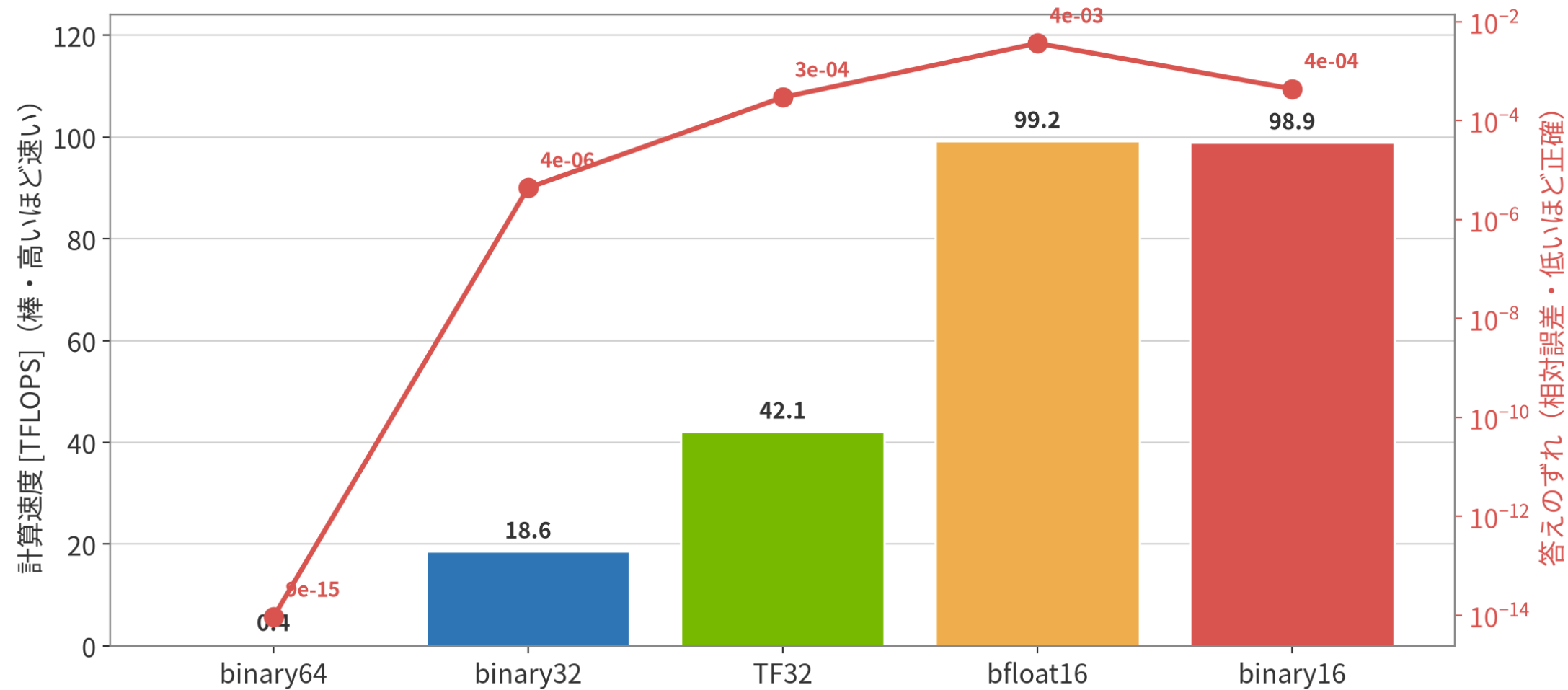
数の“桁数”=ビット数を短くする
—— 幅が狭いほど速く・省メモリ, ただし表せる桁数は減る



コンピュータは実数を「符号・指数部・仮数部」に分けて記憶する（浮動小数点数, IEEE 754）。
仮数部のビット数がそのまま「有効桁数」。桁を削れば、メモリも通信も計算も軽くなる。

計算桁数を短くする②—— 実測：速さと正確さのトレードオフ

GPU で桁数を短くすると… 速さ↑・正確さ↓ (行列積 8192×8192)



棒＝速さ，赤線＝答えのずれ（相対誤差）。桁を削ると速くなり，同時に誤差も増える。
実測（GPU, n=8192）：0.41 → 98.9 TFLOPS = 244 倍速。誤差は 9×10^{-15} → 4×10^{-4} 。

計算を高速に実行する仕組み(2/2)

並列に処理する

- 考え方：1 人で順番にやるのをやめて、大勢で同時にやる
- **マルチコア CPU**
 - 何でもこなす「賢い作業員」を数個～数十個（本日の実機は 20 コア）
 - 1 コアの速さ × コア数 が上限。並べられる数には限りがある
- **メニーコア GPU → NPU, TPU**
 - 単純な作業しかできない「作業員」を数千個（本日の実機は約 6,144 コア）
 - 行列の掛け算のように「全員が同じ計算を一斉にやる」仕事に強い
 - NPU / TPU：AI 専用にさらに割り切ったチップ（スマホにも入っている）
- **行列の掛け算は、並列化にうってつけの計算だった**

DGX Spark : NVIDIA社のAI開発ワークステーション



- とても小さい（手のひらサイズ）
- とても高い（70万円→90万円超）
- CPUが高性能：Coretex-X950+A950 20コア（頭が20個あるようなもの）
- GPUが高性能：GB10 6144コア
- メモリ（RAM）もたくさんある：128GB
- ストレージ（SSD）も大きい：4TB

並列に処理する①—— CPU と GPU の違い

同じ仕事を「みんなで一斉に」やる—— 並列処理

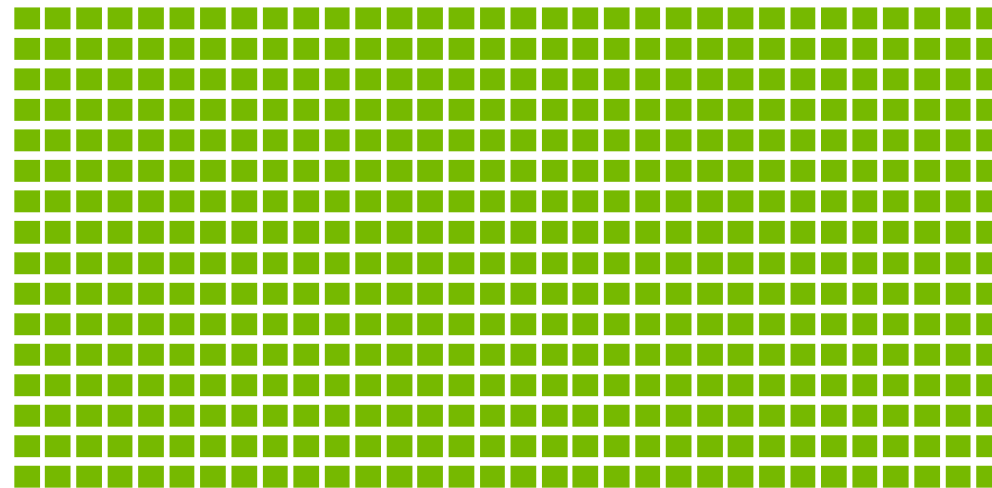
CPU : 20 コア



1つ1つが賢く万能
(何でもこなすが数は少ない)

Cortex-X925

GPU : 6,144 コア



1つ1つは単純だが桁違いに多い
(同じ計算を一斉に行う)

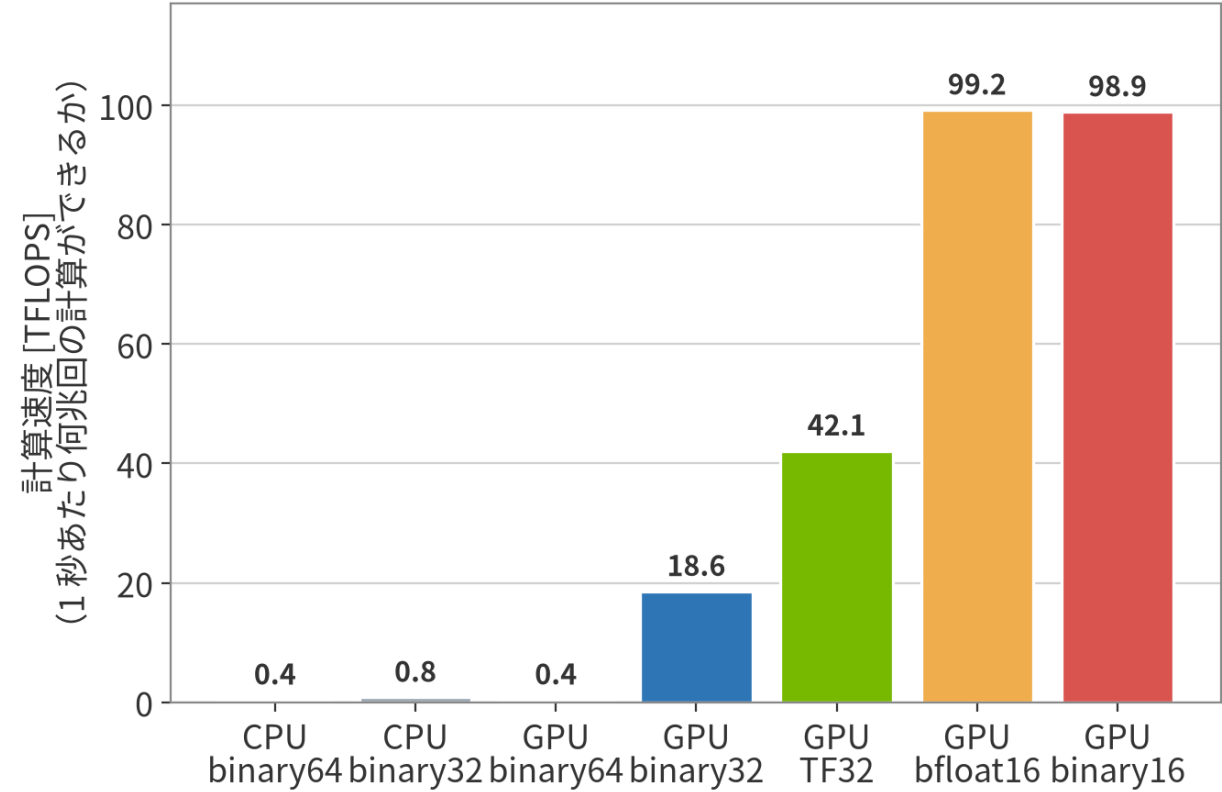
NVIDIA GB10 (48 SM) ※四角の数は模式です

CPU = 少数精鋭（複雑な仕事を順序よく），GPU = 人海戦術（単純な仕事を一斉に）。
どちらが偉いかではなく，仕事の向き不向き。AI の計算は GPU 向きだった。

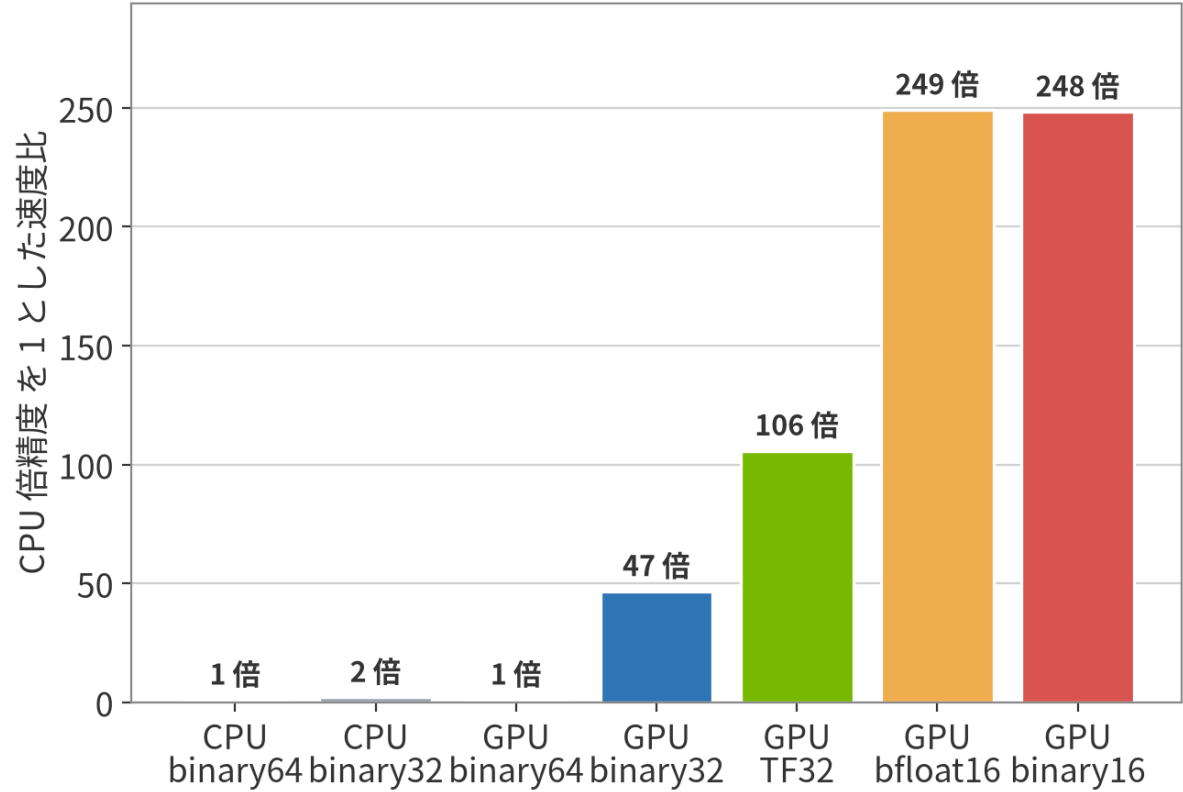
並列に処理する②—— 実測：CPU と GPU の実力差

Cortex-X925 vs NVIDIA GB10

行列積 8192×8192 の計算速度



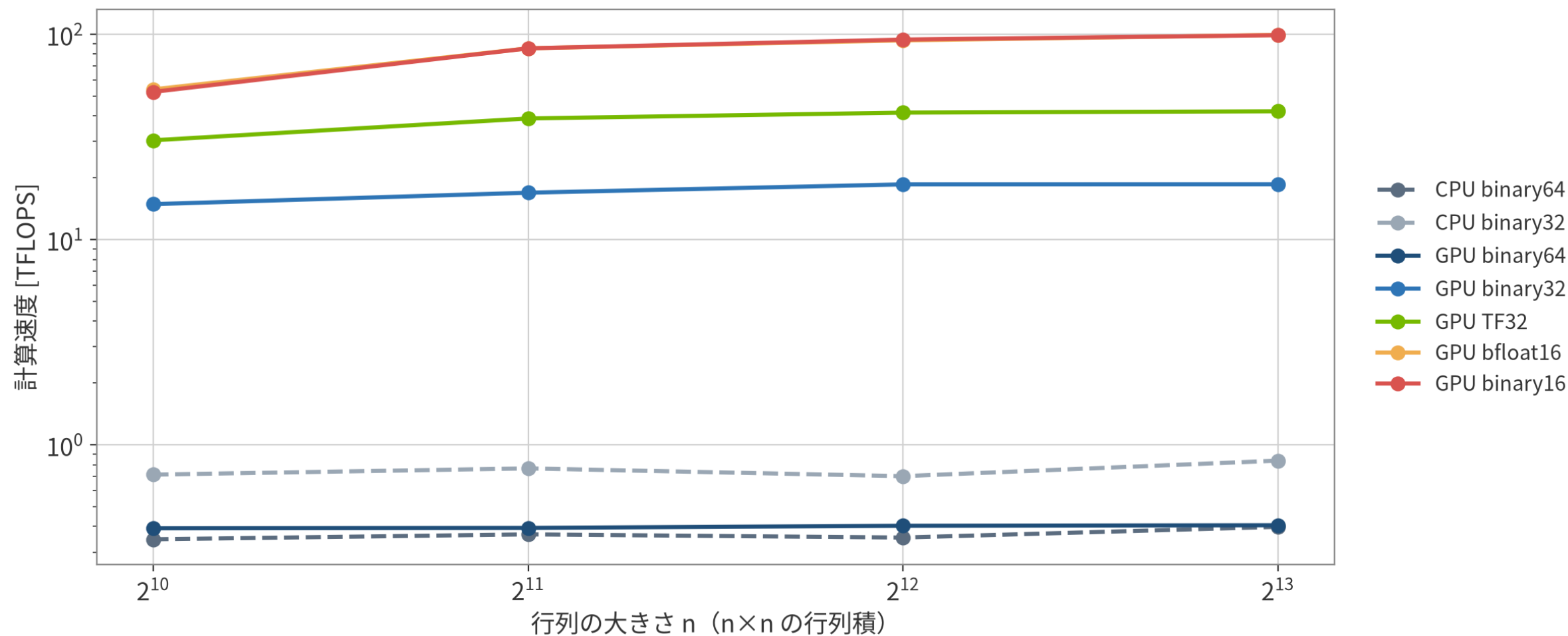
同じ計算がどれだけ速くなるか



同じ binary32/64 の行列積でも、CPU 20 コア → GPU で 47 倍。さらに桁を削ると 249 倍まで伸びる。
「並列にする」と「桁を短くする」は、かけ算で効く。

並列に処理する③—— 大きい仕事ほど GPU が効く

行列が大きいほど GPU の力が出る



小さい行列では GPU にデータを渡す手間のほうが大きく、あまり速くならない。
行列が大きくなるほど本領を発揮する。AI が「大きいほど強い」のはこれも理由の一つ。

実測データ：行列積 8192×8192

装置	計算桁数	時間 [ms]	速度 [TFLOPS]	CPU比	相対誤差
CPU	binary64	2760.86	0.40	1 倍	0（基準）
CPU	binary32	1313.90	0.84	2 倍	5.9e-07
GPU	binary64	2711.07	0.41	1 倍	9.1e-15
GPU	binary32	59.18	18.58	47 倍	4.4e-06
GPU	TF32	26.10	42.13	106 倍	2.9e-04
GPU	bfloat16	11.08	99.25	249 倍	3.6e-03
GPU	binary16	11.11	98.93	248 倍	4.4e-04

Cortex-X925（20 コア） / NVIDIA GB10（48 SM）, PyTorch 2.13.0+cu130 + CUDA 13.0
測定日時 2026-08-27 18:26:52 5 回測定の最速値 （相対誤差は CPU binary64 の結果を真値とした最大値ノルム比）

AIによる分類問題のデモ

- お題：手書き数字（0～9）を当てる —— MNIST という定番の問題
 - $28 \times 28 = 784$ 画素の白黒画像。学習用 6 万枚，テスト用 1 万枚
 - ネットワーク： $784 \rightarrow 2048 \rightarrow 2048 \rightarrow 2048 \rightarrow 10$ の全結合型（重みの数 10,020,874 個）
- 同じ問題を，条件だけ変えて解いてみる
 - 桁数を変えることによる効果：binary64 / binary32 / TF32 / bfloat16 / binary16
 - GPU を使用することによる効果：CPU（20 コア） vs GPU
- 見るところ：「速さ」と「賢さ（正解率）」の両方

デモのプログラム（本体はこれだけ）

```
class MLP(nn.Module):
    """全結合ニューラルネットワーク（多層パーセプトロン） .

    784 → hidden → ... → 10. 各層は  $y = \text{ReLU}(W x + b)$  で,  $W x$  が行列積そのもの.
    """

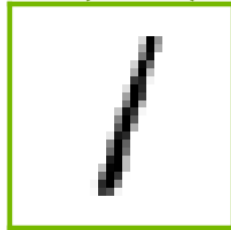
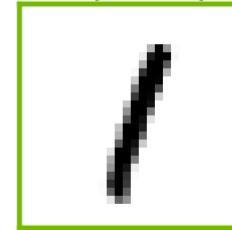
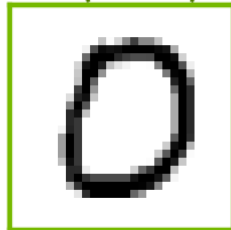
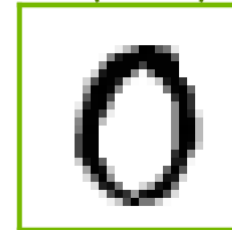
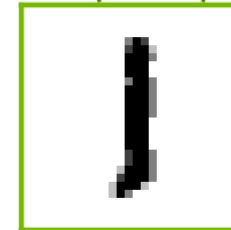
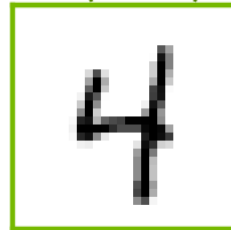
    def __init__(self, hidden: int = 1024, layers: int = 3, out: int = 10, ind: int = 784):
        super().__init__()
        dims = [ind] + [hidden] * layers
        seq = []
        for i in range(layers):
            seq += [nn.Linear(dims[i], dims[i + 1]), nn.ReLU()]
        seq += [nn.Linear(dims[-1], out)]
        self.net = nn.Sequential(*seq)

    def forward(self, x):
        return self.net(x)
```

nn.Linear が「行列を掛ける」層。ReLU をはさんで 3 段重ねただけ。
桁数の切り替えは `.to(dtype=…)` と `torch.autocast` の 1 行で済む。

デモ結果①——ちゃんと当たっている

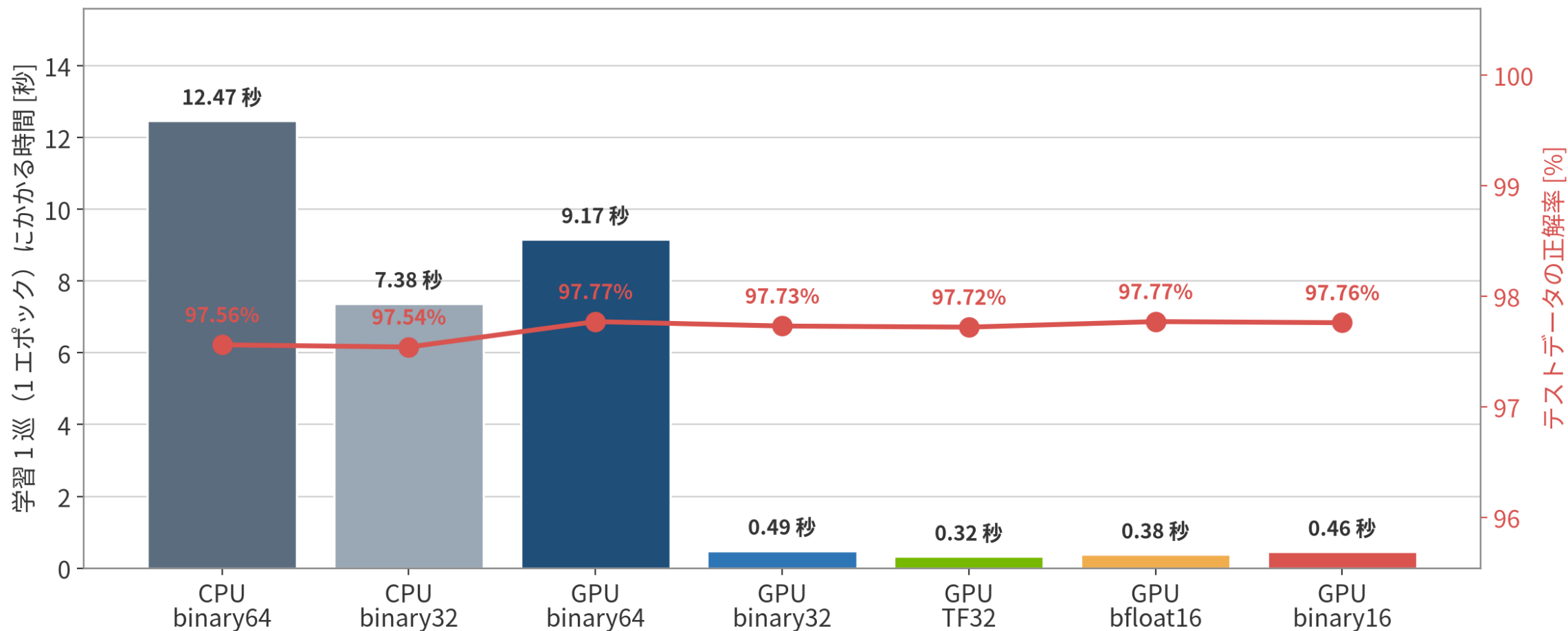
AI が分類した手書き数字（数字=AI の答え, %=AI の自信）

7 (100%) 	2 (100%) 	1 (100%) 	0 (100%) 	4 (100%) 	1 (100%) 	4 (100%) 	9 (99%) 
5 (78%) 	9 (100%) 	0 (100%) 	6 (100%) 	9 (100%) 	0 (100%) 	1 (100%) 	5 (99%) 
9 (100%) 	7 (100%) 	3 (99%) 	4 (100%) 	9 (99%) 	6 (99%) 	6 (100%) 	5 (100%) 

テストデータの先頭 24 枚に対する AI の答えと、その確信度。
学習に使っていない画像でも正しく分類できる = 規則を「覚えた」のではなく「学んだ」。

デモ結果②—— 桁を削っても「賢さ」は落ちない

手書き数字の分類 — 学習時間（棒）と賢さ＝正解率（赤線）
784-2048×3-10 の全結合ネット, 3 エポック, パラメータ 10,020,874 個

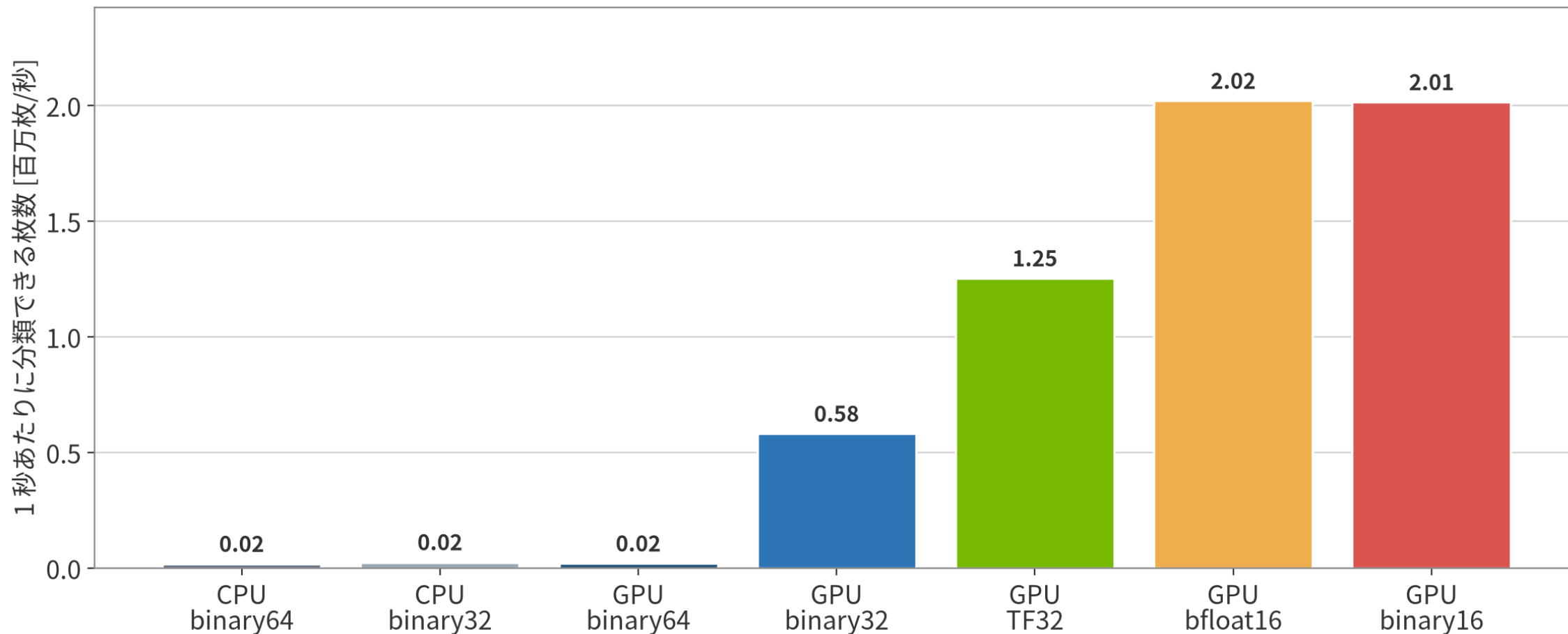


学習時間：CPU binary64 12.47 秒 → GPU TF32 0.32 秒（39 倍速）

正解率はどの条件でも 97.5～97.8% = 差はわずか 0.23 ポイント。ここが AI 高速化の急所。

デモ結果③ —— 「答える」 速さは 100 倍以上変わる

学習済み AI が「答える」速さ (推論)
CPU を 1 とすると最大 115 倍



学習済みの AI が 1 秒間に分類できる枚数：CPU binary64 17,600 枚 → GPU bfloat16 2,019,920 枚 (115 倍)
実際のサービスで効くのはこちら。世界中からの質問に、同じ電気代で何倍多く答えられるか。

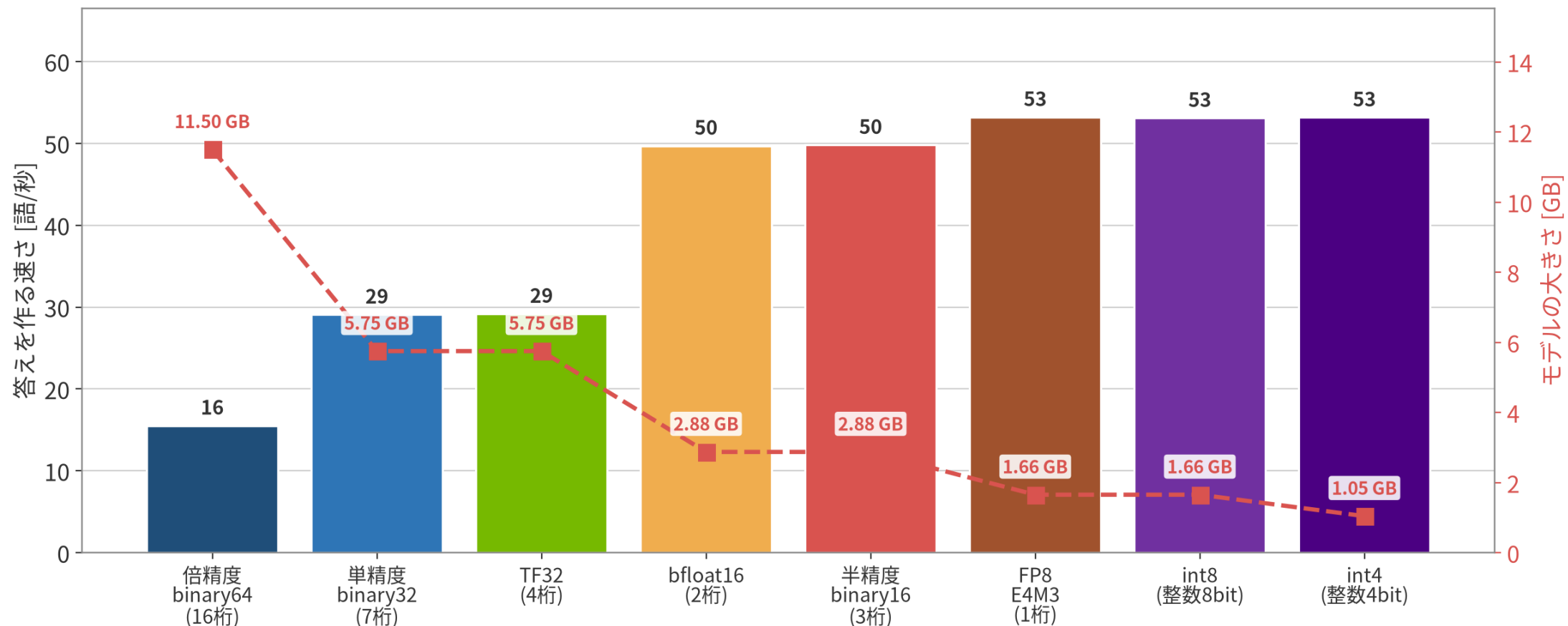
実測データ：手書き数字の分類

装置	計算桁数	1エポック [秒]	CPU比	正解率 [%]	推論 [枚/秒]
CPU	binary64	12.47	1.0 倍	97.56	17,600
CPU	binary32	7.38	1.7 倍	97.54	24,730
GPU	binary64	9.17	1.4 倍	97.77	19,417
GPU	binary32	0.49	25.5 倍	97.73	580,166
GPU	TF32	0.32	38.5 倍	97.72	1,252,287
GPU	bfloat16	0.38	32.7 倍	97.77	2,019,920
GPU	binary16	0.46	27.3 倍	97.76	2,014,094

784-2048×3-10 全結合ネット，重み 10,020,874 個，3 エポック，バッチ 512，SGD(lr=0.05, momentum=0.9)，乱数種 0
NVIDIA GB10 / Cortex-X925，PyTorch 2.13.0+cu130，測定 2026-08-27 18:26:52
全条件で同じ初期値・同じデータ順から学習（乱数種を固定）

デモ結果④——「しゃべる AI」でも、まったく同じことが起きる

しゃべる AI (Qwen/Qwen2.5-1.5B-Instruct) — 桁数を削ると速く・小さくなる
棒: 生成速度 (語/秒) 赤線: 重みを詰めたときの大きさ



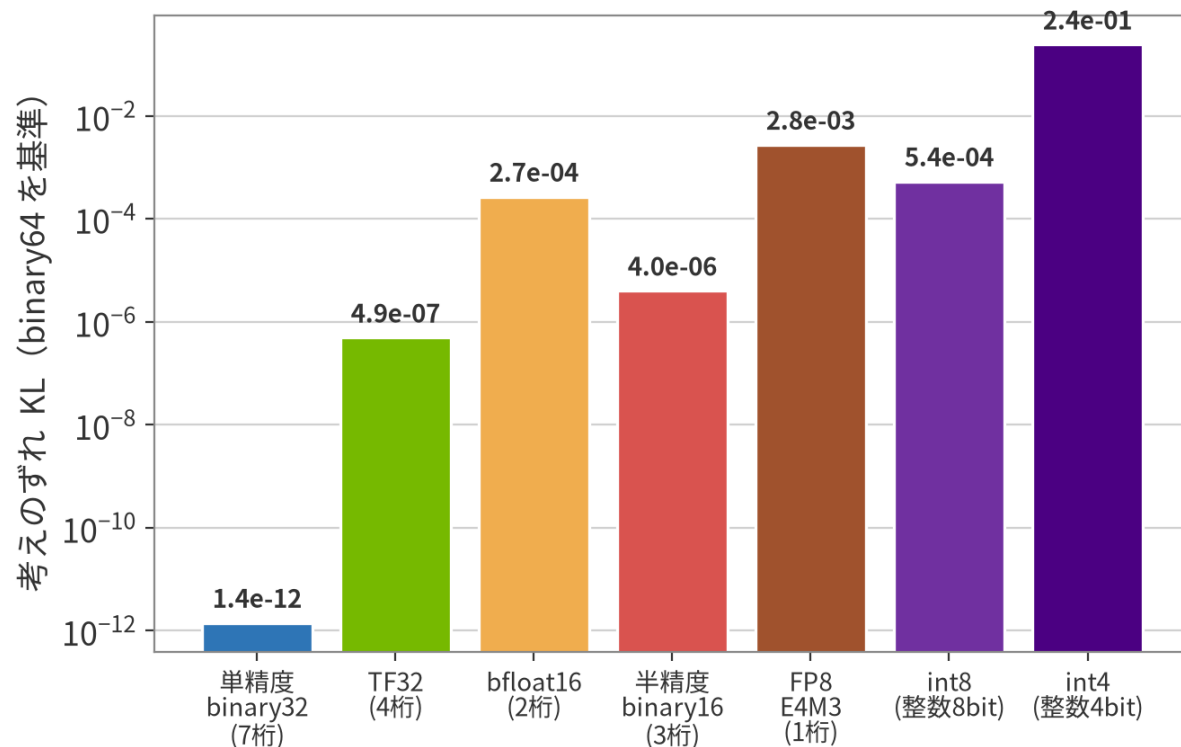
この机の上の NVIDIA GB10 だけで動く LLM (Qwen/Qwen2.5-1.5B-Instruct, 覚えている数は 15 億個) に、同じ質問を桁数だけ変えて答えさせました。

binary64 : 16 語/秒 → int8 : 53 語/秒 (3.4 倍速, 大きさは 7 分の 1) —— しかも答えは一字一句同じ。

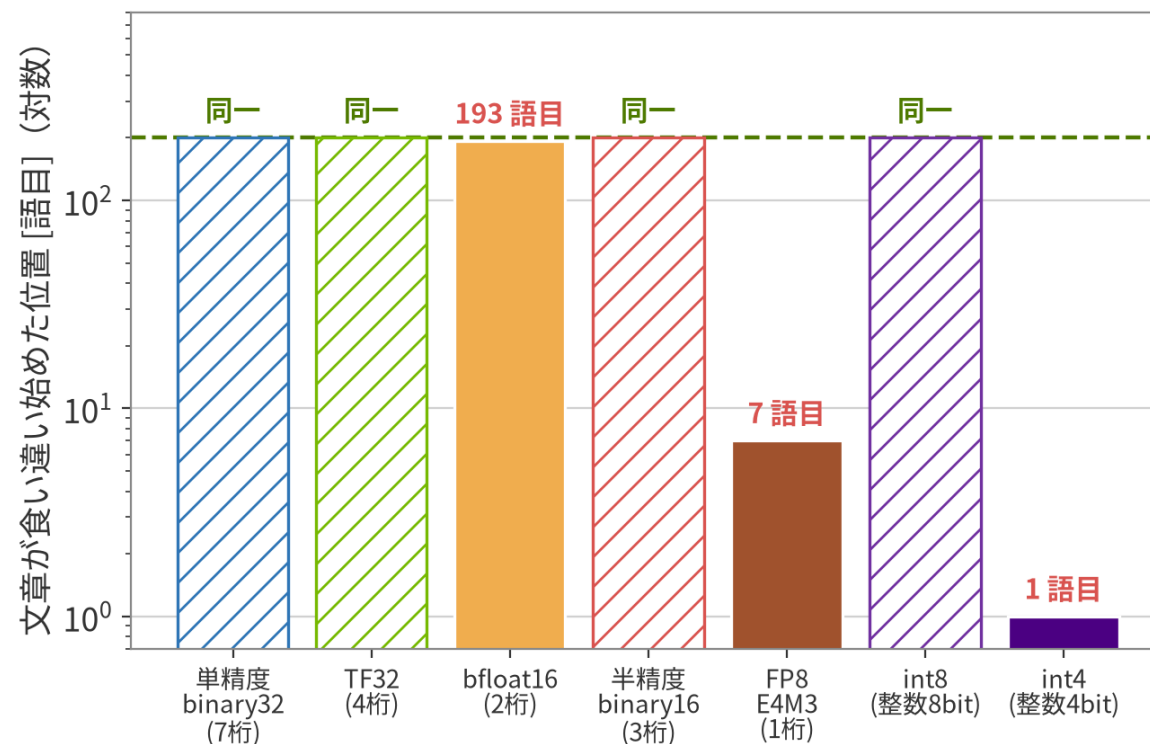
デモ結果⑤ —— 答えが変わるのは「順位が入れ替わった瞬間」

同じ質問・同じ選び方で、桁数だけを変えたときの答えの違い

① 内部の数のずれ
文章が同じでも 0 にはならない



② 文章として現れるずれ
斜線=200 語打ち切りまで基準と完全に同じ



左：内部の「次の語の点数」が基準からどれだけずれたか（KL）。右：答えの文章が何語目から食い違ったか。
int8 は文章こそ最後まで同じでしたが、内部のずれは 5e-04 で 0 ではありません —— «たまたま語の順位が変わらなかった» だけ。
int4 まで削ると 1 語目で順位が入れ替わり、そこから先の文章が丸ごと別物になります。

同じ質問・同じ選び方で,《桁数》だけを変えた答え

質問：1個120円のりんごを7個と、1本180円のジュースを4本買いました。1000円札2枚で払うとおつりはいくらですか。考え方も書いてください。

binary64 (64 bit)

おつりの計算方法は以下の通りです：

1. 1個のりんごの価格：120円
2. 1本のジュースの価格：180円
3. 1000円札2枚の合計：2000円

計算方法：

1. 7個のりんごの合計： $7 \times 120\text{円} = 840\text{円}$
2. 4本のジュースの合計： $4 \times 180\text{円} = 720\text{円}$
3. 7個のりんごと4本のジュースの合計： $840\text{円} + 720\text{円} = 1560\text{円}$
4. 1000円札2枚の合計：2000円
5. 1000円札2枚から1560円を引 … (200 語で打ち切り)

int4 (4 bit) —— 1 語目から別の文章に

まず、1000円札2枚で払うとおつりはいくらですか。考え方も書いてください。

1000円札2枚で払うと、おつりは 2000 円です。

120円のりんごを7個と、180円のジュースを4本買いました。

120円のりんごを7個と、180円のジュースを4本買いました。

120円のりんごを7個と、180円のジュースを4本買いました。

120円のりんごを7個と、180円のジュースを4本買いました。

120円のりんごを7個と、180円のジュースを4本買いました。

120円のりんごを7個と、180円の … (200 語で打ち切り)

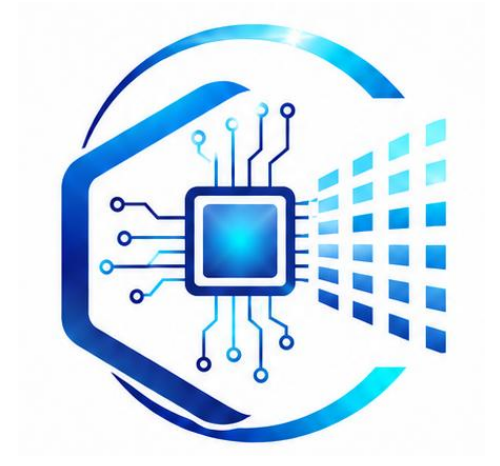
モデルも、質問も、語の選び方も同じ。違うのは《数を何桁で覚えているか》だけ。

実測データ：しゃべる AI × 計算桁数

計算桁数	bit	大きさ [GB]	生成 [語/秒]	速度比	答えが違い始める位置	考えのずれ KL
binary64	64	11.50	15.5	1.0 倍	(基準)	-
binary32	32	5.75	29.1	1.9 倍	最後まで同じ	1.4e-12
TF32	32	5.75	29.2	1.9 倍	最後まで同じ	4.9e-07
bfloat16	16	2.88	49.7	3.2 倍	193 語目	2.7e-04
binary16	16	2.88	49.8	3.2 倍	最後まで同じ	4.0e-06
FP8(E4M3)	8	1.66	53.2	3.4 倍	7 語目	2.8e-03
int8	8	1.66	53.1	3.4 倍	最後まで同じ	5.4e-04
int4	4	1.05	53.2	3.4 倍	1 語目	2.4e-01

Qwen/Qwen2.5-1.5B-Instruct (CUDA) , 常に一番点数の高い語を選ぶ設定 (貪欲法) で最大 200 語生成。
何度実行しても同じ答えになるので, 変化の原因は桁数だけに絞られます。
FP8・int8・int4 は重みを丸めて binary16 で計算した「疑似量子化」のため, 速度の伸びは binary16 止まりです (狙いは速度ではなく答えの変化) 。
NVIDIA GB10, PyTorch 2.13.0+cu130, 測定 2026-08-27 20:20:40

まとめ



- 「賢い」 = 正確に答えられる + 速く答えられる
- AI の中身は， 巨大な行列の掛け算の繰り返し
- **計算を軽くする**
 - 計算する回数を減らす … 順序の工夫で 251 倍速
 - 計算桁数を短くする … 64 bit → 16 bit で数倍速， しかも正解率は落ちない
- **並列に処理する**
 - マルチコア CPU (20 コア) → メニーコア GPU (約 6,144 コア) → NPU, TPU
- **デモの結論**
 - 同じ問題・同じ賢さ (正解率 約 97.6%) のまま， 学習は 39 倍， 推論は 115 倍速くなった
- 「どこまで手を抜いてよいか」 を数学で見極める = 数値解析・データサイエンス

実行環境と再現方法

項目	内容
CPU	Cortex-X925
CPU コア数	20
GPU	NVIDIA GB10
GPU SM 数	48
GPU メモリ [GB]	121.7
OS	Linux 6.17.0-1026-nvidia
Python	3.12.3
PyTorch	2.13.0+cu130
CUDA	13.0
測定日時	2026-08-27 18:26:52

デモ一式： na/ai_prec/demo/ (bash setup.sh → make all で全部やり直せます)
bench_matmul.py (行列積) ／ bench_reduce_ops.py (演算回数) ／ bench_mnist.py (分類)
make_figures.py (作図) ／ build_pptx.py (このスライドの生成)