第１３回計算科学技術特論B(2026)
GPUと低精度計算
NVIDIA
大友広幸

2026年7月9日（木）13:00－14:30

主催：高度情報科学技術研究機構(RIST)
次世代HPC・AI研究開発支援センター(HAIRDESC)

共催：東京大学物性研究所

後援：理化学研究所計算科学研究センター、
計算物質科学人材育成コンソーシアム(PCoMS)

GPU と低精度計算

Hiroyuki Ootomo

Take-home message: 低精度演算の活用がますます重要になる

- 近年の GPU では、FP64 よりも FP16、FP8 などの低精度演算の性能が大きく伸びている
- 対応する数値形式も世代ごとに増え、計算精度を選べる場面が広がっている
- データの表現に使うビット数を減らせば、高速な演算器を使え、またメモリ使用量や通信量も削減できる
- 一方、科学技術計算では必要な精度を保たなければならない

本講義で伝えたいこと
最新 GPU の性能を引き出すには、低精度演算を計算のどこに、どのように組み込むかが重要になる。

必要な精度を保ちながら、計算全体を速くすることが目標。

NVIDIA.

今日の話

- 浮動小数点形式の復習と最近のトレンド
- 低精度・混合精度計算の利点と注意点
- NVIDIA が提供する低精度・混合精度対応ライブラリ
- 研究紹介
- 残っている課題

サンプルコード

`https://github.com/enp1s0/cst26b_lp`

Contents

C 言語の `float`/`double` 型

一般的な CPU/GPU 環境での対応

C/C++ の型	呼び方	およその桁数
`float`	単精度 / FP32	7 桁
`double`	倍精度 / FP64	16 桁

```
float x = 1.0f; // FP32
double y = 1.0; // FP64
```

- 実数を有限のビット数で表すため、表現できる値は飛び飛びになる
- `double` は `float` より多くのビットを使い、近い値を細かく区別できる

この講義でいう**計算精度**は、数値をどの程度細かく表現し、計算できるかを指す。

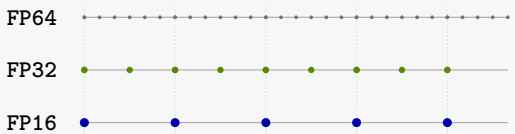
NVIDIA

精度を下げると、近い値を区別できる間隔が広がる

1 の次に表せる値

形式	1 の次の値	1 との間隔
FP64	1.0000000000000002	2^{-52}
FP32	1.0000001192092896	2^{-23}
FP16	1.0009765625	2^{-10}

精度は表現の細かさ

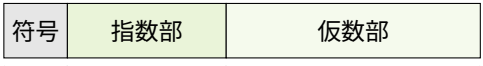


概念図: 同じ指数部範囲の場合

- 1 のすぐ隣に表せる数を比べると、形式ごとの違いが見える
- FP16 では、1 付近でも約 0.001 より細かい違いは表せない
- 表せない値は、近くの表現可能な値へ丸められる

低精度になるほど、同じ範囲でも表現できる値の間隔が広くなる。

浮動小数点数は符号・指数部・仮数部で表す



符号 $\times$ 仮数 $\times 2^{\text{指数}}$

- **符号**: 正の数か負の数か
- **指数部**（exponent）: どの程度大きな値や小さな値まで表せるか
- **仮数部**（mantissa）: 近い二つの値をどこまで区別できるか

ビット数を減らすと、表せる範囲が狭くなるか、値の刻みが粗くなる。多くの場合はその両方が起こる。

十進数の 6.02×10^{23} 表記を、コンピュータでは二進数の $1.m \times 2^{e}$ で表している。

浮動小数点形式の基本

形式	全ビット	符号	指数	仮数	大まかな特徴
FP64 (IEEE 754)	64	1	11	52	科学技術計算の標準、約 16 桁
FP32 (IEEE 754)	32	1	8	23	約 7 桁
TF32	32 保存	1	8	10	FP32 指数部、FP16 仮数部
BF16	16	1	8	7	FP32 指数部、仮数部は短い
FP16 (IEEE 754)	16	1	5	10	仮数部は BF16 より長いが指数部が短い
FP8 E4M3 (OCP)	8	1	4	3	範囲より精度寄り
FP8 E5M2 (OCP)	8	1	5	2	精度より範囲寄り

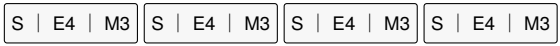
表記: 指数部 x bit、仮数部 y bit のフォーマットを ExMy と表す。

同じ 16 bit や 8 bit でも FP16 と BF16、E5M2 と E4M3 では特徴が異なる。必要な桁数だけでなく、入力値の大きさも見て形式を選ぶ。

出典・参考: IEEE 754; NVIDIA Hopper Architecture In-Depth

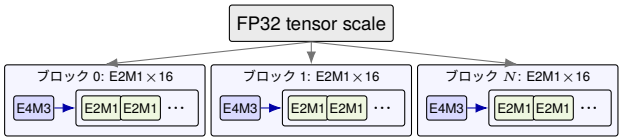
Block-scaled floating-point 系

通常の FP8



MXFP8

NVFP4



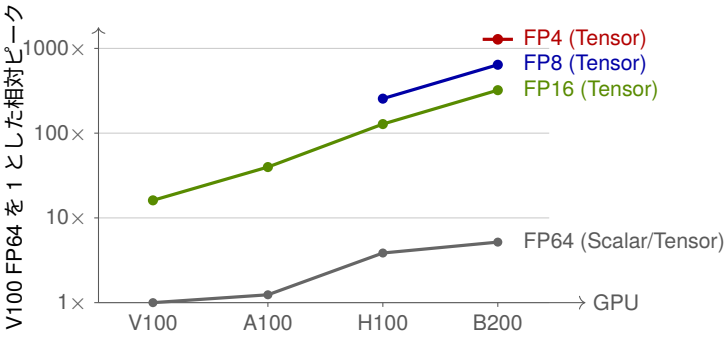
形式	要素	共有 scale 値	ブロックサイズ
FP8 (OCP)	E4M3/E5M2	なし	1
MXFP8	E4M3/E5M2	E8M0 / block	32
NVFP4	E2M1	E4M3 / block FP32 / tensor	16

スケールの役割
ブロック内の要素をまとめて拡大・縮小し、各要素のビット数が少ない場合でも、表現範囲を広くできる。
NVFP4 では、さらに一つの FP32 scale を複数のブロックで共有する。

共有スケールによって指数部部分の情報圧縮を実現できる一方、同じブロックに大きさの極端に異なる値があると表現しにくい。

出典・参考: OCP Microscaling Formats (MX) Specification v1.0; NVIDIA Blackwell Architecture Technical Brief

GPU 世代が進むほど FP64 と低精度演算の性能差が広がっている



- 一般に、計算精度と計算速度はトレードオフ
- `FP64` は世代が進んでも伸び方が比較的緩やか
- 世代を経るごとに低精度演算の性能が向上し、対応する形式も増えている
- `FP8` は H100 世代、`FP4` は Blackwell 世代でサポート開始

出典・参考: NVIDIA V100/A100/H100/Blackwell GPU architecture materials; NVIDIA Tensor Cores page。各世代の代表的な dense peak（sparsity なし）を V100 FP64 比で表示

Contents

低精度・混合精度計算では速さと精度を同時に設計する

- **低精度計算**: より少ないビット数の数値形式で表し、より高速な演算器を利用可能とする。またメモリ使用量や通信量の削減も可能となる
- **混合精度計算**: 計算する場所ごとに精度を変え、必要な精度を保つ
- GPU では Tensor コアや低精度演算器によって低精度・混合精度演算のピーク性能が伸びている
- 科学技術計算では「速いが不正確」では使えず、**誤差を制御する設計**が必要

低精度・混合精度演算で考えたいこと

1. どこをどれくらい低精度にしてよいか
2. 精度が落ちたらどう回復するか

低精度・混合精度計算の何が嬉しいか

計算のボトルネックがメモリバンド幅でも演算でも高速化の可能性がある

メモリバンド幅ネック

- 演算に対してデータ要求が多く、計算の性能がメモリアクセス処理で律速されている状態
- 例: ベクトル加算（演算 n、データ要求 $3n$）

演算ネック

- データ要求に対して演算が多く、計算の性能が演算で律速されている状態
- 例: $n \times n$ 行列積（演算は約 $2n^3$、データ要求は約 $3n^2$）

最終結果の精度が許すのであれば:

- メモリバンド幅ネックなら、メモリ上での保存精度を下げる
- 演算ネックなら、低精度のより速い演算器で計算する

ことで、計算を高速化できる可能性がある。

まずは計算のボトルネックを特定することが大切

NVIDIA

いろいろな低精度・混合精度計算

保存形式だけを小さくする

- 元々 `FP64` などの配列を `FP16`、`BF16`、`FP8` に変換して保存
- メモリ容量と帯域を節約
- 読み出してから `FP64` などに戻して計算する

演算精度を下げる

- Tensor コアなどの高速な低精度・混合精度演算器の性能を生かしやすい
- 計算のボトルネックがメモリバンド幅ではなく演算の場合に効果が大きい
- 例: 深層学習

精度を回復する

- 低精度を速く処理
- 補正計算・再計算で精度を回復する
- 精度が必要な科学技術計算で特に重要
- 例: 行列積 Emulation

Contents

低精度化で起こりうる精度面の問題を知る

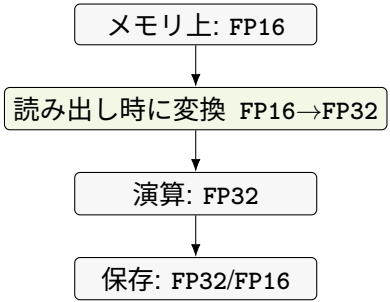
低精度で起きること

- 近い値が同じ値として扱われやすくなる
- 小さな更新が大きな値に埋もれやすくなる
- 値域を外れ `inf` や 0 になりやすくなる

形式	見える桁の感覚	注意
FP64	約 15 桁	科学計算でよく使う基準
FP32	約 7 桁	倍精度より指数部も仮数部も小
TF32	約 3–4 桁	Tensor コアでのみ利用可能
BF16	約 2–3 桁	値域が FP32
FP16	約 3–4 桁	値域が FP32 より狭い
FP8	約 1 桁	値の大きさをそろえて使う

まずは、表せる値の範囲、実際に低精度化した場合の計算結果内の `NaN/inf` の有無、高精度結果との差を確認する。

計算のボトルネックを考える



考えることの例:

- 保存だけ低精度: バンド幅節約が主目的
- 演算は高精度で行っても、依然としてメモリバンド幅ネック ⇒ 演算も低精度にしてもあまり速くならない可能性大
- 高精度演算の分、計算結果の誤差を小さくすることが可能

計算のボトルネックを考えることで、計算精度・計算性能トレードオフを改善する

性能の出る GPU 資源の利用法・プログラミング手法の確認

自分で CUDA カーネルを書く場合、単に低精度変数を使うだけだと性能が十分に上がらない可能性がある。まず、自分の計算で低精度資源を使えるかを確認

FP16 演算器

- FP16 の型は `half`
- Scalar 演算の計算性能を十分得るには、`half2` を用いた SIMD 演算が必要

```
const float f0 = 1.f, f1 = 4.6f, f2 = 4.8f;

const half2 v2 = __float22half2_rn(f0, f1);
const half2 s2 = __float2half2_rn(f2);

const half2 r2 = __fmul2(s2, v2); // SIMD (f16x2)
```

メモリアクセスの効率化

- GPU では、128 bit 分など、いくつかのデータをまとめてメモリ読み書きするのが好ましい
- 低精度変数を利用する際は、128 bit 分などのアクセスができるよう、プログラムの書き換えを行ったほうが良いことがある

カーネル内での Tensor コアの利用

後ほど

NVIDIA

低精度計算を試す際に確認すると良いこと

どれくらい正確ならよいか
必要な桁数や、許容できる結果の差を先に決める。

値は表せる範囲に入るか
入力と途中の値の最小値・最大値を見る。

失敗を見つけられるか
保存量、高精度結果との差など、成否を判断する基準を決める。

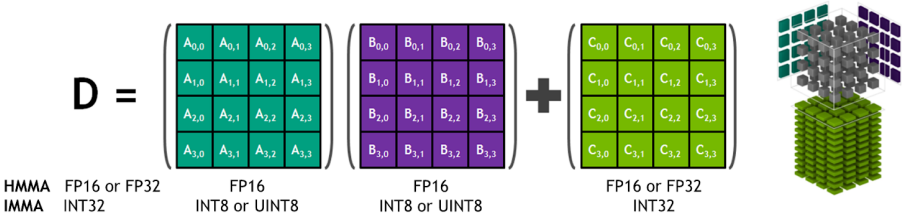
ホットスポットか
時間のかかっている部分の高速化ほど、全体の計算の高速化につながる。

うまく GPU の低精度資源を使えるか
どんな計算でも速くなるわけではない

最初からすべてを低精度にせず、作業用配列や近似計算など一部分から試す。

Contents

Tensor コアとは



- 混合精度行列積和演算回路
- 入力行列が低精度
- 最近のほとんどの NVIDIA GPU で利用可能
- 深層学習での行列積の需要から、Volta 世代で初めて搭載
- A100 からは FP64 の Tensor コアも

Tensor コア自体は小さな行列積を計算する回路。大きな行列積は部分行列積に分割して Tensor コアに入力する

Tensor コアとは

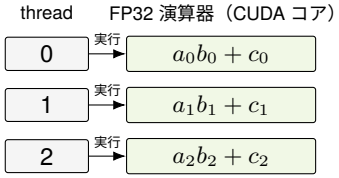
- ハードウェアとしては、SM 内に FP32 演算器などとともに実装
- 複数の thread が協調して Tensor コアでの計算を実行
  - 基本は 1 warp
  - Hopper 以降では 4 warp 協調もサポート

復習:

- thread: GPU で動く最小の処理単位
- warp: 32 個の thread をまとめた実行単位

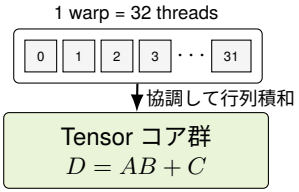
CUDA コアは汎用演算、Tensor コアは行列積和に特化

CUDA コア: thread ごとに演算



- スカラー・ベクトル演算を柔軟に実行
- 条件分岐、添字計算、変換にも使う
- 要素ごとの処理や不規則的な計算が得意

Tensor コア: warp で行列積和



- 固定形状の小さな $D = AB + C$ を実行
- 多数の積和を一つの命令で処理

Tensor コアを使うカーネルでも、アドレス計算や制御は通常の CUDA 命令で行う。行列積にまとめられる部分だけを Tensor コアへ任せる。

Tensor コアを使うには

Tensor コア対応のライブラリを使う

- 例: cuBLAS、cuDNN、cuTENSOR 等
- 自分の計算がライブラリが提供する計算に合致するのであれば、性能最適化済みの実装が利用可能

カーネル内で Tensor コア API を呼ぶ

- WMMA (Warp Matrix Multiply and Accumulate) API
- サイズ固定の小さな行列積和を Tensor コアで行うための C++ API

Tensor コアを有効に活用するには、どの方法でもボトルネックが演算となる十分大きな行列積の形が出てくる必要がある

- ダメな例: ベクトルの内積も行列積の特殊な場合ではあるけれど、Tensor コアを用いたところで速くはならない

NVIDIA

cuBLAS から Tensor コアを使う

単精度行列積

```
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            m, n, k,
            &alpha,
            A, lda,
            B, ldb,
            &beta,
            C, ldc);
```

内部で FP16 Tensor コアを用いる場合

```
cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N,
        m, n, k,
        &alpha,
        A, CUDA_R_32F, lda,
        B, CUDA_R_32F, ldb,
        &beta,
        C, CUDA_R_32F, ldc,
        CUBLAS_COMPUTE_32F_FAST_16F,
        CUBLAS_GEMM_DEFAULT);
```

- Tensor コアを利用することで計算時間は短縮されるが、内部で入力行列を `FP16` へ変換するため計算精度は劣化する
- `CUBLAS_COMPUTE_32F_FAST_16F` を変更することで、内部計算を切り替え可能

まずは cuBLAS の関数・引数を変え、基準の FP32 計算と実行時間・結果を比較する。

カーネル内から Tensor コアをつかう: WMMA API

WMMA (Warp Matrix Multiply and Accumulate) APIを用いることでカーネル関数内から Tensor コアでの計算を実行可能

1. `fragment` 変数を用意
2. `load_matrix_sync` で行列を読み込む
3. `mma_sync` で $D = A \times B + C$ を計算
4. `store_matrix_sync` で結果を書き戻す

WMMA API で計算できる行列積和
例: $\mathbf{A} \cdot \mathbf{B} + \mathbf{C}$, where

- $\mathbf{A}, \mathbf{B}$: 16×16 FP16 行列
- $\mathbf{C}$: 16×16 FP32 行列

fragment とは
1 warp = 32 threads で行列を協調保持する際の、1 thread が持つ行列ブロック

出典・参考: NVIDIA CUDA C++ Programming Guide, Warp Matrix Functions

WMMA API 最小サンプル

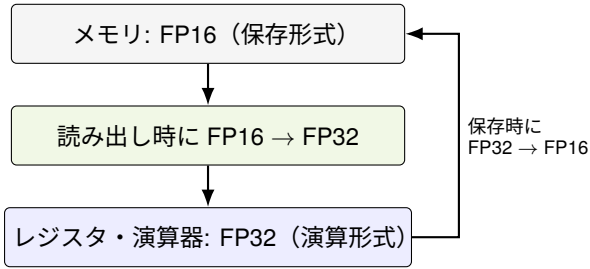
```cpp
#include <cuda_fp16.h>
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_16x16(const half *A, const half *B, float *C) {
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

  // 行列Cの全要素を0
  wmma::fill_fragment(c_frag, 0.0f);

  // 行列A, Bの読み込み
  wmma::load_matrix_sync(a_frag, A, 16);
  wmma::load_matrix_sync(b_frag, B, 16);

  // 行列積和 C = A*B+C
  wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

  // 結果の書き出し
  wmma::store_matrix_sync(C, c_frag, 16, wmma::mem_row_major);
}
```

Contents

保存に使うビット数を減らすとメモリ使用量や転送量を減らせる

- 変数や配列を低精度形式で保存する
- 計算するときだけ高精度へ変換する方法もある
- 計算方法を大きく変えずに試せる場合が多い
- 同じ時間でより多くの要素を読み込める

演算器を変えなくても、データを小さくするだけで得られる利点がある。



研究紹介: Custom 8-bit floating point value format for reducing shared memory bank conflict in approximate nearest neighbor search

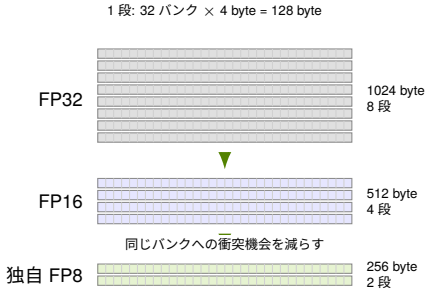
研究概要
低精度変数を使うことで Shared memory の bank conflict の回数を減らし、計算性能を改善する。保存形式は 8 bit、計算精度は FP32 の混合精度

用途に合わせた 8-bit 形式

- 計算特性上、符号部を省略
- `E5M3` と `E4M4` を設計
- FP32 との変換は少数の命令で可能

NVIDIA cuVS で提供中

256 要素の参照表を 32 バンクに配置



精度（recall）はやや落ちるが、FP32・FP16 より高いスループットを得られる。

出典・参考: Ootomo and Naruse, Custom 8-bit floating point value format for reducing shared memory bank conflict in approximate nearest neighbor search, 2023, arXiv:2301.06672

Contents

反復的に計算精度を改善する例

方程式 $3x = 10$　　**最初の近似解** $x_0 = 3.3$　　**途中計算精度** 2 桁　　**残差計算精度** 4 桁

1. 代入してずれを調べる

$$3x_0 = 3 \times 3.3 = 9.9$$

左辺は、右辺の 10 より 0.1 小さい。

$$r_0 = 10 - 9.9 = 0.1$$

この 0.1 が残差。

2. 補正更新量 z の計算

x を z 増やすと、左辺 $3x$ は $3z$ 増える。左辺を 0.1 増やすには、

$$3z = r_0$$

$$z = \frac{r_0}{3} \approx 0.033$$

とすればよい。

3. 更新してもう一度確かめる

$$x_1 = x_0 + z = 3.333$$

$$3x_1 = 9.999$$

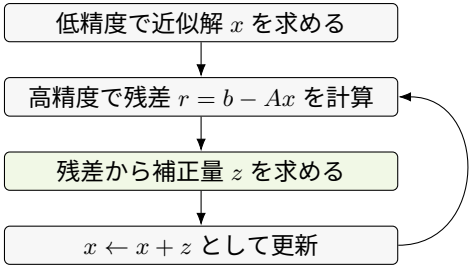
新しい残差は、

$$r_1 = 10 - 9.999 = 0.001$$

まで小さくなる。

NVIDIA

反復改良は補正を繰り返して解の精度を高める

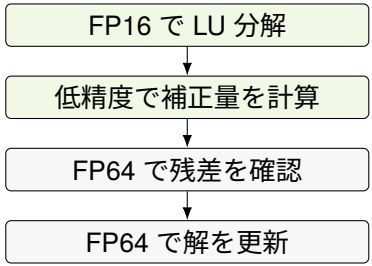


- 近似解 x が $Ax = b$ をどれだけ満たしていないかを表すのが**残差** r
- 重い計算は低精度で速く行う
- 答えの確認は高精度で行う
- 残差が十分小さくなるまで繰り返す

低精度で近似解を求め、高精度でずれを測りながら補正する。

研究紹介: Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers

- **問題**: FP64 だけで線形方程式を解くと時間がかかる
- **低精度にする部分**: 計算量の大きい LU 分解と補正計算
- **高精度に残す部分**: 残差の計算と解の更新
- FP16 Tensor コアを使いながら、最終的には FP64 と同程度の精度を目指す



出典・参考: Haidar et al., Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers, SC18, 2018, DOI

cuSOLVER で混合精度の反復改良は利用可能

- `cusolverDnIRSXgesv()` は密行列の $Ax = b$ を数値求解する関数
- 入力・出力データの精度を指定する
- 内部計算で用いる最低精度も指定する
- 低精度で収束しない場合は、高精度計算へ戻せる

```
cusolverDnIRSParams_t params;
cusolverDnIRSInfos_t  infos;
cusolverDnIRSParamsCreate(&params);
cusolverDnIRSInfosCreate(&infos);

cusolverDnIRSParamsSetSolverPrecisions(
    params, CUSOLVER_R_64F, CUSOLVER_R_TF32);
cusolverDnIRSParamsSetRefinementSolver(
    params, CUSOLVER_IRS_REFINE_GMRES);

cusolverDnIRSXgesv_bufferSize(handle, params, n, nrhs, &lwork);
cudaMalloc(&work, lwork);
cusolverDnIRSXgesv(handle, params, infos,
                   n, nrhs, dA, lda, dB, ldb, dX, ldx,
                   work, lwork, dinfo);
```

出典・参考: NVIDIA cuSOLVER documentation: IRS solver, `cusolverDnIRSXgesv()`

Contents

行列積精度エミュレーション

Tensor コアによる混合精度行列積を複数回利用して、高精度の行列積の結果を模倣する

基本アイデア

1. 入力の高精度行列をいくつかの低精度行列に分割
2. 低精度行列同士の積を Tensor コアで計算。結果は高精度で得る
3. 行列積の計算結果を統合

手法

1. TF32x3, BF16x9 などの FP32 行列積エミュレーション
2. Ozaki scheme
3. Ozaki scheme II

Ozaki scheme と Ozaki scheme II の詳細は、この後の尾崎先生の講義で扱う。

NVIDIA

研究紹介: Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance

- **問題**: FP32 入力をそのまま FP16/TF32 へ変換すると情報が失われる
- **基本アイデア**: 失われた情報をもう一つの FP16/TF32 で保持し、計算結果を補正する
- FP32 の精度を保ったまま、FP32 の理論性能以上の計算性能を達成

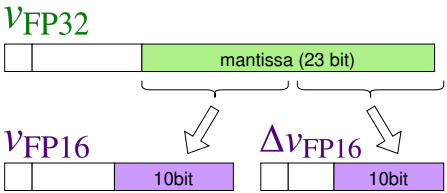
簡単な例

1.234×5.678 を 2 桁精度入力の混合精度積和を使って 4 桁精度で計算

$$
\begin{aligned}
1.234 \times 5.678 &= (1.2 + 3.4 \times 10^{-2}) \times (5.6 + 7.8 \times 10^{-2}) \\
&= (1.2 \times 5.6) + \underbrace{(1.2 \times 7.8 + 3.4 \times 5.6) \times 10^{-2} + (3.4 \times 7.8) \times 10^{-4}}_{\text{Correction term}} \\
&= 6.72 + 28.4 \times 10^{-2} + \underbrace{26.52 \times 10^{-4}}_{\text{can be negligible}} \\
&\sim 7.007
\end{aligned}
$$

出典・参考: Ootomo and Yokota, Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance, 2022, arXiv:2203.03341

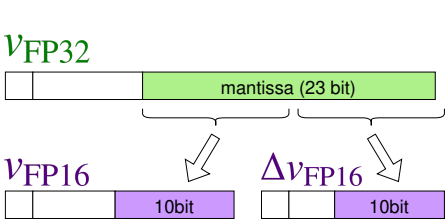
SGEMM emulation on Tensor Cores ($\mathbf{C}_{\text{FP32}} \leftarrow \mathbf{A}_{\text{FP32}} \cdot \mathbf{B}_{\text{FP32}}$)

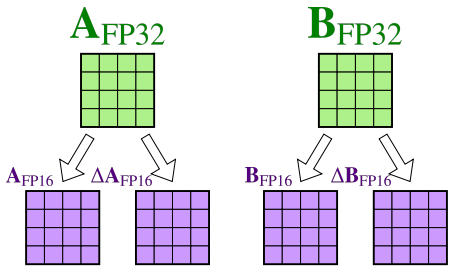


$$v_{\text{FP32}} \approx v_{\text{FP16}} + \Delta v_{\text{FP16}}$$

NVIDIA.

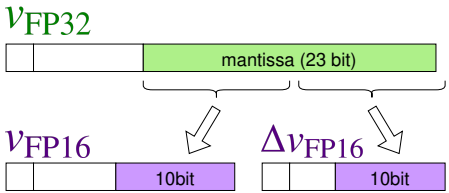
SGEMM emulation on Tensor Cores ($\mathbf{C}_{\text{FP32}} \leftarrow \mathbf{A}_{\text{FP32}} \cdot \mathbf{B}_{\text{FP32}}$)





NVIDIA

SGEMM emulation on Tensor Cores ($C_{FP32} \leftarrow A_{FP32} \cdot B_{FP32}$)

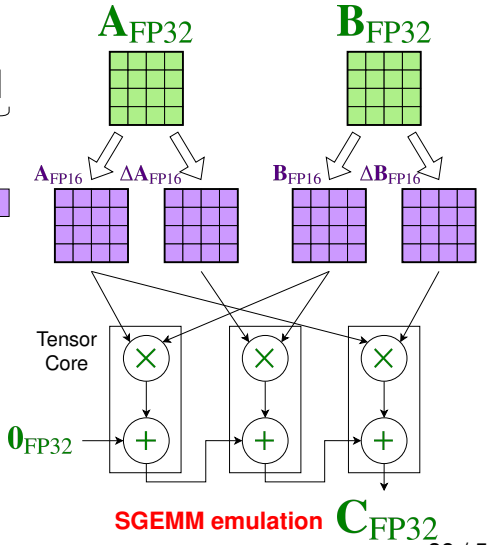


$$v_{FP32} \approx v_{FP16} + \Delta v_{FP16}$$

Scheme

$$C_{FP32} \approx A_{FP16} \bullet B_{FP16} + \Delta A_{FP16} \bullet B_{FP16} + A_{FP16} \bullet \Delta B_{FP16}$$

with avoidance of RZ inside Tensor Cores



FP16・TF32・BF16 では速度と値域のバランスが違う

方式	基本的な作り方	特徴
`FP16x3`	主成分と補正成分を FP16 で計算	高速だが、表せる値の範囲が狭い
`TF32x3`	主成分と補正成分を TF32 で計算	FP32 とほとんど同じ範囲を取り扱える
`BF16x9`	FP32 を 3 成分ずつに分け、9 個の積を計算	入力行列の仮数部を完全に保持できる。cuBLAS で利用可

cuBLAS で試す

CUDA 12.9 以降の対応 GPU では、FP32 emulation mode を指定すると BF16x9 方式を利用できる。通常の SGEMM と同じような手順で試せる。

cuBLAS で BF16x9 を使う

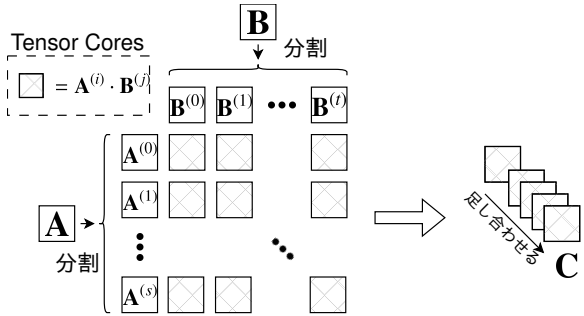
```
cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N,
        m, n, k,
        &alpha,
        A, CUDA_R_32F, lda,
        B, CUDA_R_32F, ldb,
        &beta,
        C, CUDA_R_32F, ldc,
        CUBLAS_COMPUTE_32F_EMULATED_16BFX9,
        CUBLAS_GEMM_DEFAULT);
```

出典・参考: Ootomo and Yokota, 2022; NVIDIA cuBLAS documentation, Floating Point Emulation

研究紹介: DGEMM on Integer Matrix Multiplication Unit

ここでは考え方だけを紹介する。Ozaki scheme と Ozaki scheme II の詳細は、この後の尾崎先生の講義で扱う。

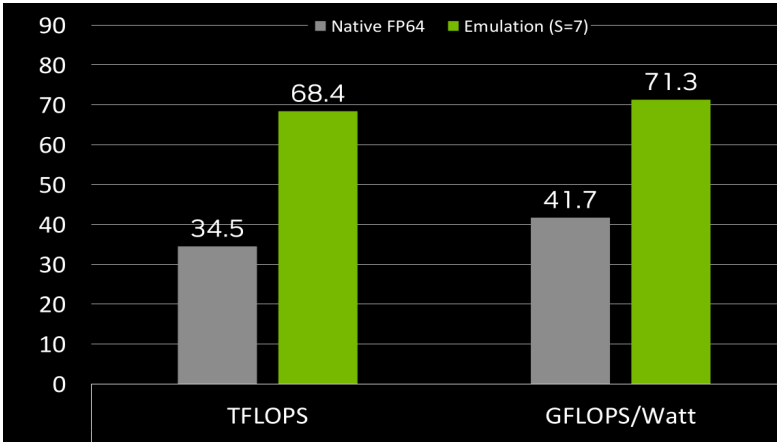
- 入力行列の FP64 を固定小数に変換し、8 bit 整数に分割する
- 分割された Int8 行列の行列積を Int8 Tensor コアで計算する
- 部分積を Int32 で足し合わせ、最後に FP64 へ戻す



出典・参考: Ootomo, Ozaki and Yokota, DGEMM on integer matrix multiplication unit, 2024, DOI

どれくらい速いのか？

- B200 GPU 上で、TOP500 に用いられる計算 HPL を実行
- 実質行列積の計算性能のランキング



- 計算性能で 2 倍、電力効率で 1.7 倍

cuBLAS で Ozaki scheme を使う

引数で指定する

```
cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N,
        m, n, k,
        &alpha,
        A, CUDA_R_64F, lda,
        B, CUDA_R_64F, ldb,
        &beta,
        C, CUDA_R_64F, ldc,
        CUBLAS_COMPUTE_64F_EMULATED_FIXEDPOINT,
        CUBLAS_GEMM_DEFAULT);
```

環境変数で指定する

```
export CUBLAS_EMULATE_DOUBLE_PRECISION=1
```

裏で cuBLAS の倍精度行列積が呼ばれていれば、C/C++ 以外からでも利用可能

```
import cupy as cp

n = 4096
a = cp.arange(n*n, dtype=cp.float64).reshape(n,n)
b = cp.eye(n, dtype=cp.float64)

c = cp.dot(a, b)
```

対応環境: CUDA 13.0 Update 2 以降の対応 GPU

ドキュメント: https://docs.nvidia.com/cuda/cublas/

Contents

実行時の精度選択

低精度・混合精度計算は計算精度と計算速度のトレードオフ。実行時によしなに計算精度を選択することはできる？

1. 入力値の範囲を測る
↓
2. 内部計算精度や分割数を選ぶ

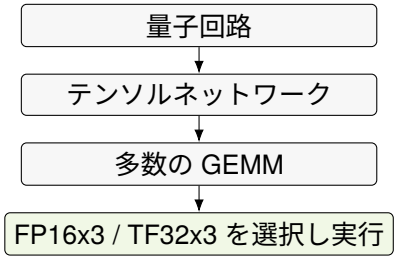
- 入力によって安全な精度は変わる
- GPU 世代によって速い形式も変わる
- 最初から一つに固定せず、実行時に選びたい
- 必要なときは fallback を行う

fallback: 低精度で問題が起きそうなときに高精度へ戻す安全策。

NVIDIA

研究紹介: Quantum Circuit Simulation by SGEMM Emulation on Tensor Cores and Automatic Precision Selection

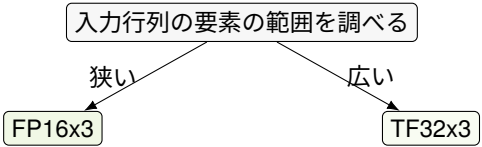
- **問題**: 量子回路シミュレーションで多数の行列積を高速化したい
- **基本アイデア**: SGEMM emulation によって Tensor コアで計算する
- 行列ごとに値の範囲を調べ、FP16x3 または TF32x3 を選ぶ



出典・参考: Ootomo et al., Quantum Circuit Simulation by SGEMM Emulation on Tensor Cores and Automatic Precision Selection, 2023, arXiv:2303.08989

行列の値の範囲が狭ければ FP16x3、広ければ TF32x3 を選ぶ

- FP16x3 は高速だが、表せる値の範囲が狭い
- TF32x3 は FP32 に近い範囲を持つ
- 行列の最小値・最大値から、値の広がりを調べる
- FP16 の範囲に収まる行列では FP16x3、収まらない行列では TF32x3 を使う



同じプログラムでも、行列ごとに適した低精度形式が違う。

自動選択には高精度へ戻れる安全策が必要になる

- 少ない計算量で入力値の範囲を調べる
- 条件に合う低精度形式を選ぶ
- 計算結果が要求を満たすか確認する
- 条件を満たさない場合は高精度へ戻す
- 選んだ精度を記録し、再現できるようにする

目指す使い方

- ユーザは必要な精度を指定
- ライブラリが形式を選択
- 失敗時は自動で再計算
- GPU が変わっても使える

ユーザがすべての型を手作業で選ばなくても、安全な範囲で低精度演算器を利用できることが理想

Contents

実用的な低精度計算には、正しさ・速さ・使いやすさが必要になる

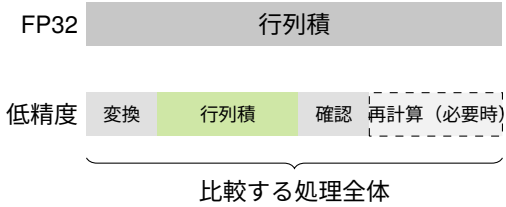
正しい結果	実際に速い	簡単に使える
結果の確認	変換・補正のコスト	自動選択と再計算

低精度演算器は部品の一つ。結果の検証やデータ変換まで含めた計算方法として設計する必要がある。

NVIDIA.

演算器が速くても、前処理や再計算によって高速化の効果が小さくなる

- 値の範囲の調査、型変換、スケーリングにも時間がかかる
- 結果の確認や、高精度での再計算にも時間がかかる
- 小さな行列やメモリ律速の計算では、Tensor コアの高い演算性能を生かしにくい
- カーネル単体ではなく、処理全体の時間とデータ移動量を測る



$$T_{\text{total}} = T_{\text{prepare}} + T_{\text{compute}} + T_{\text{check}} + T_{\text{retry}}$$

評価すべきなのはピーク性能ではなく、計算が完了するまでの時間（time-to-solution）。

要求精度と GPU 固有の実装を分け、世代間の移植性を高める

- GPU 世代ごとに、高速な数値形式や Tensor コアで計算効率の良い行列積の形
- 積の精度、accumulation の精度、スケーリング方法も形式ごとに異なる
- 同じ問題でも、最適な形式や行列の分け方は GPU によって変わる
- ライブラリが GPU に応じて自動でいい感じのものを選べる形が望ましい

アプリケーションと実装の役割分担

- アプリ: 必要な精度を指定
- ライブラリ: 形式とカーネルを選択
- 実行時: GPU の機能を確認

Contents

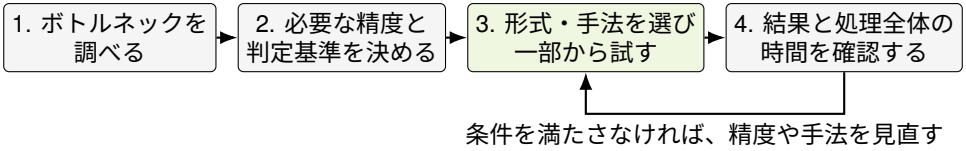
ボトルネックと必要な精度に応じて、低精度の使い方を選ぶ

保存精度を下げる	低精度で演算する	必要な精度を回復する
メモリ・通信がボトルネック • 配列を FP16、BF16、FP8 などで保存 • 読み出した後の演算は高精度にもできる • 容量とデータ移動量を減らす • 保存時に失った情報は戻らない	**演算がボトルネック** • まずは自分の計算が低精度計算で高い性能を得られる計算かを考える • cuBLAS や WMMA API で Tensor コアを利用	**演算がボトルネック** • 反復改良で解を補正 • 複数の混合精度積で高精度計算をエミュレート • まずは cuBLAS や cuSOLVER で試す

型を一律に置き換えるのではなく、低精度化可能な部分を考えながら・探しながら行う

NVIDIA

低精度計算は、ボトルネックを調べ、結果と全体時間を確認しながら導入する



結果について確認すること

- 高精度の基準結果との差
- 残差・保存量・収束履歴
- 値域、`NaN`、`inf`

性能について確認すること

- カーネルではなく処理全体の時間
- 変換・スケーリング・結果確認の時間
- fallback した回数と時間

低精度演算器が速くなるほど、必要な精度を保ちながら、計算のどこに組み込むかが重要になる。