第１４回計算科学技術特論B(2026)
HPC環境における行列積のエミュレーション（１）

芝浦工業大学
尾崎克久

2026年7月16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)

自己紹介

タイ・バンコク郊外にて
良い子は真似しないでください

- 芝浦工業大学
- システム理工学部
- 数理科学課程

- 研究内容:
  - 高信頼数値計算
  - 精度保証付き数値計算
  - 丸め誤差解析
  - 数値線形代数

講義概要（7/16, 7/23）

- 7/16
  - 導入的内容
  - 尾崎スキームIに関する話題（CPU版・GPU版）

- 7/23
  - 尾崎スキームIIに関する話題（CPU版・GPU版）
  - スキームの性質
  - 尾崎スキームの応用
  - 現状の課題について

浮動小数点数

$s * m * 2^e$

- IEEE 754 の2進フォーマット：
  - FP16 ：1 bitの符号部, 5 bitの指数部, **10** bitの仮数部
  - FP32 ：1 bitの符号部, 8 bitの指数部, **23** bitの仮数部
  - FP64 ：1 bitの符号部, 11 bitの指数部, **52** bitの仮数部
  - **FP128**：1 bitの符号部, 15 bitの指数部, **112** bitの仮数部

（原著的には binary**という）

（正規化数ならば暗黙の１も使える)

現状ハードウェアサポートはほぼ？なし

- IEEE 754以外の2進フォーマット
  - FP4, FP6, FP8（通常2種類, いや3種類？）, TF32, BF16, …
- 仮数部が長ければ精度に期待
- 指数部が短ければオーバーフローやアンダーフローの心配

概要

Multiword arithmeticや
多倍長精度計算には
本日は触れない点，ご了承ください

尾崎スキームI（※１）：

- 行列積のためのエラーフリー変換（スプリットによる手法（※２））
- **チューニング済**の行列積のコードを使うだけの方法
- エミュレーション技法と呼ばれる

※１英語ではOzaki-I schemeと呼ばれることが多くなった
※２ネイティブはスライシングということもある（方法次第だと思う）.

何に使えるのか？

- その１：CPU上で高精度計算を高速化したい
- その２：GPU上で倍精度演算**相当**の計算を高速化したい

尾崎スキームIの紹介の前に

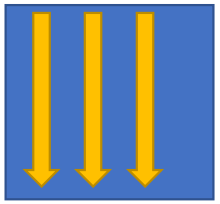
行列積のコードはベタに書かない（１）

- 何も考えずに書く場合（サイズn，正方行列の積）

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

```
for (i = 0; i < n; ++i) { // i行
    for (j = 0; j < n; ++j) { // j列
        for (k = 0; k < n; ++k) {
            C[i + n*j] += A[i + k*n] * B[k + n*j];
        }
    }
}
```

行列を１次元配列（列優先）で保存しています．



行列積のコードはベタに書かない（２）

- **メモリの連続アクセス**くらいは知っている
- 以下は列優先保存

```
for (j = 0; j < n; ++j) { j列
    for (k = 0; k < n; ++k) {
        for (i = 0; i < n; ++i) { // i行
            C[i + n*j] += A[i + k*n] * B[k + n*j];
        }
    }
}
```

２次元配列を使うときは注意
C言語とFortranで優先方向が異なる

行列積のコードはベタに書かない（３）

- マルチコアなんだからOpenMPくらいは知っている
- #pragma omp parallel for private(i, k)
- for (j = 0; j < n; ++j) {　　// j 列
- for (k = 0; k < n; ++k) {
- for (i = 0; i < n; ++i) { // i 行
- C[i + n*j] += A[i + k*n] * B[k + n*j];
- }
- }
- }

<omp.h>をインクルード
/openmpを入れてコンパイル

行列積のコードはベタに書かない（４）

FP64の正方行列の積に対する計算時間（秒）

行列のサイズ	（１）	（２）	（２）＋（３）	MATLAB
1024	1.81	0.48	0.095	0.022
2048	44.3	4.43	0.83	0.15
4096	475.0	13.3	3.99	1.13

- ループアンローリングは？
- FMAは使われているっけ？（a*b+cを一度に計算）
- ベクトル化しないと！（AVXとかを有効活用）
- キャッシュレベルでブロッキングしなきゃ
- レジスタブロッキング（レジスタ圧の調整）をしなきゃ

無料で高速な行列積関数は提供されている

計算機環境：
Core i7-8550U
MATLAB R2025a
oneAPI使用

行列積のコードはベタに書かない（５）

- 性能を測る指標：
- Floating-point Operations per Second (FLOPS)
- １秒間に何回の計算ができたか？

- 行列積は$(2n-1)n^2 = 2n^3-n^2$回の計算回数

表：先ほどの計算時間からGFLOPSを算出したもの

行列のサイズ	（1）	（2）	（2）＋（3）	MATLAB
1024	1.02	3.88	19.6	84.6
2048	0.33	3.36	17.9	99.3
4096	0.25	8.96	29.8	105

数多くありますので，全部列挙は無理です

参考：行列積が高速！（その１）

- MATLAB：
  行列演算を中心に科学技術計算・可視化・アルゴリズム開発を統合的に行える商用計算環境
- Julia：
  高性能数値計算を目的に設計され、動的言語の書きやすさとC/Fortran並みの実行性能を両立する科学技術計算向けプログラミング言語
- Scilab・ GNU Octave :
  数値計算と科学技術計算のための，MATLAB互換を志向したオープンソースの計算環境
- MKL:
  Intel CPU向けに高度に最適化された，BLAS・LAPACK・FFTなどを提供する高性能数値計算ライブラリ

数多くありますので，全部列挙は無理です

参考：行列積が高速！（その２）

- OpenBLAS:
  BLASおよび一部LAPACKを高性能に実装したオープンソース数値線形代数ライブラリ
- BLIS:
  高性能な行列演算を提供するための，ハードウェア最適化指向の線形代数ライブラリ
- SLATE:
  分散メモリ並列環境向けに設計された，最新アーキテクチャ対応の高性能線形代数ライブラリ
- CUDA:
  NVIDIAのGPUプログラミング環境，cuBLAS・cuBLASLtを使う

現代では？

- **ひと昔前**は自分で行列積のチューニングされたコードを書くのは至難の業だった（と思う）
  （これが尾崎スキームの発見につながる）

- **現代では**Claude, Codexなどと３０分くらい議論を続けると高速行列積のコードは結構書けてしまう・・・
  - BLIS流の手法を試してみて，レジスタ圧はどうなの？など・・・
  - ブロックサイズも自動で試してくれますし・・

誤差が起きない計算とは？

無誤差で計算できるとは？

- 最も簡単な例として整数演算を考える
- 行列積なので内積を，内積なので「積」と「和」のみ考える

- まずは10進数の例：
  107 * 212 = 22684

- 3桁の数と3桁の数の積は6桁みたいに，
  x桁の数どうしの積は2x桁

- 10+11+16+21+77+19+56+44+33+89 = 376
  2桁の数を10項たしたら3桁みたいに，
  x桁の数をn項足したら**x+log10 n**桁で十分（安全をみれば切り上げ）

エラーフリーに計算できるとは？

- 最も簡単に整数演算を考える
- 行列積なので，積と和のみ考える
- 次に2進数の例：

- 3ビットの数と3ビットの数の積は6ビットみたいに，
  xビットの数どうしの積は2xビット

- 2ビットの数を16項たしたら6ビットみたいに，
  xビットの数をn項足したら**$x+\log_2 n$**ビット（安全をみれば切り上げ）

エラーフリーに計算できるとは？

- 入力をxビット，積で2xビット，総和で$\log_2 n$の繰り上がり
- $2x+\log_2 n$ビットの数になる可能性

- つまりxビットの数に対して$\mathrm{ceil}(2x+\log_2 n)$ビット保存できれば正確に計算結果を保持できる

- 一方で使用できる型（FP16, FP32, FP64など）は固定されている

- 逆にyビットを保持できる環境であれば，xが**$\mathrm{floor}((y-\log_2 n)/2)$**ビットであれば良い．

入力を制限するということ

（あくまで「ワーストなケースでも大丈夫」を考えている）

尾崎スキームI

原点：K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump,
“Error-Free Transformations of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and Its Applications,”
Numerical Algorithms, Vol. 59, No. 1, pp. 95–118, 2012.
DOI: 10.1007/s11075-011-9478-1.

ひとこと：私が尾崎スキームと言い出したわけではない
海外では単にOzakiと呼ばれることもあるが・・

エラーフリー行列積の考え方

- ２つの行列 A と B の積を考える(例：FP64),
- 以下の分割を考える
- $A = A_1 + A_2 + \cdots + A_k$,
- $B = B_1 + B_2 + \cdots + B_l$.
- A_i と B_j の要素はFP64.
- $A_i * B_j$, $1 \le i \le k, 1 \le j \le l$ が**無誤差で計算できる**
- $AB = (A_1 + A_2 + \cdots + A_k)(B_1 + B_2 + \cdots + B_l)$

  $= A_1B_1 + A_1B_2 + A_2B_1 + \ldots + A_kB_l$

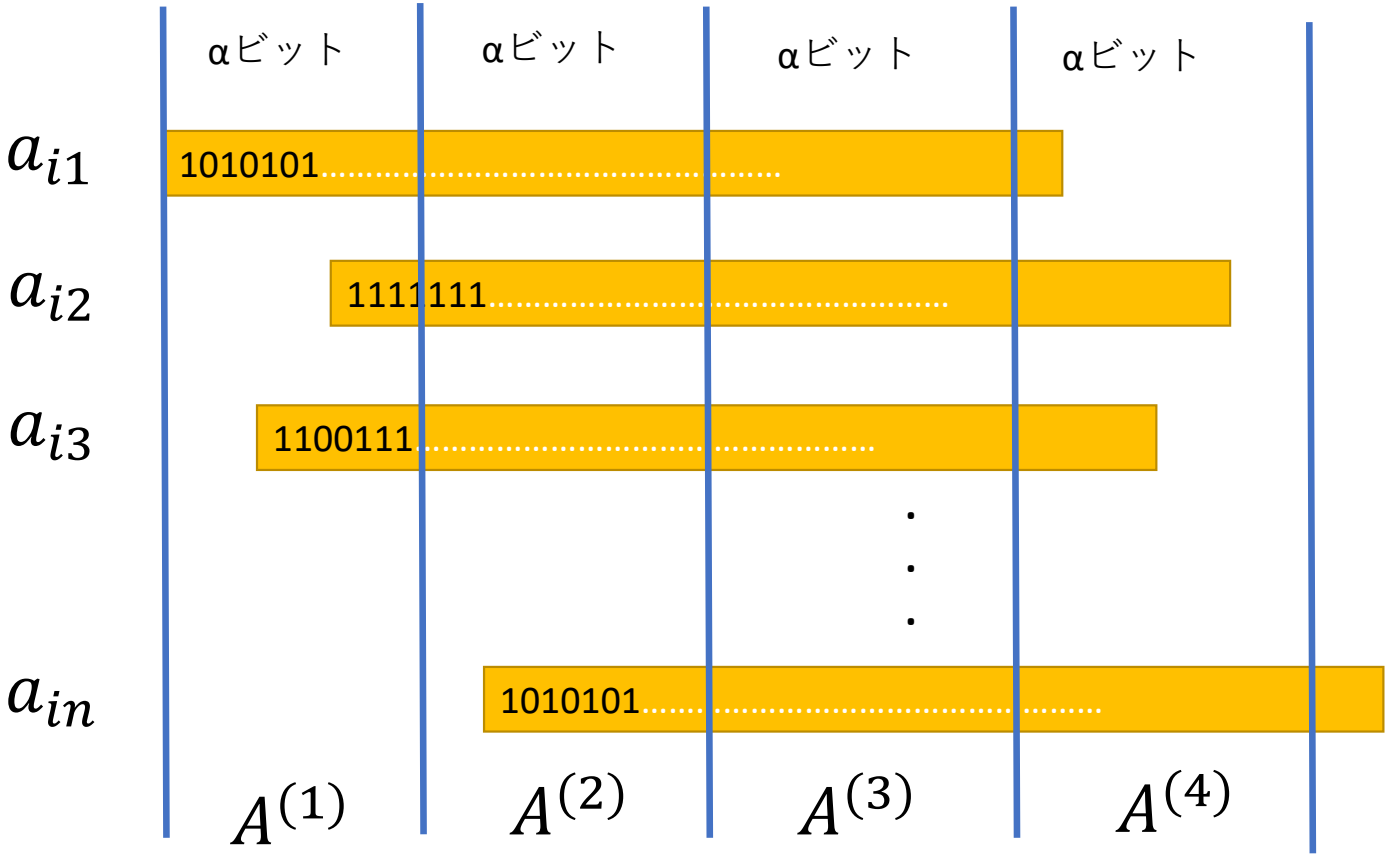
k 個の行列,
l 個の行列

最適化された計算をそのまま使う

AB は kl 個の和に誤差なく変換される

行列積のルーチンに特殊なこと（分割統治法）などが使われていないこと

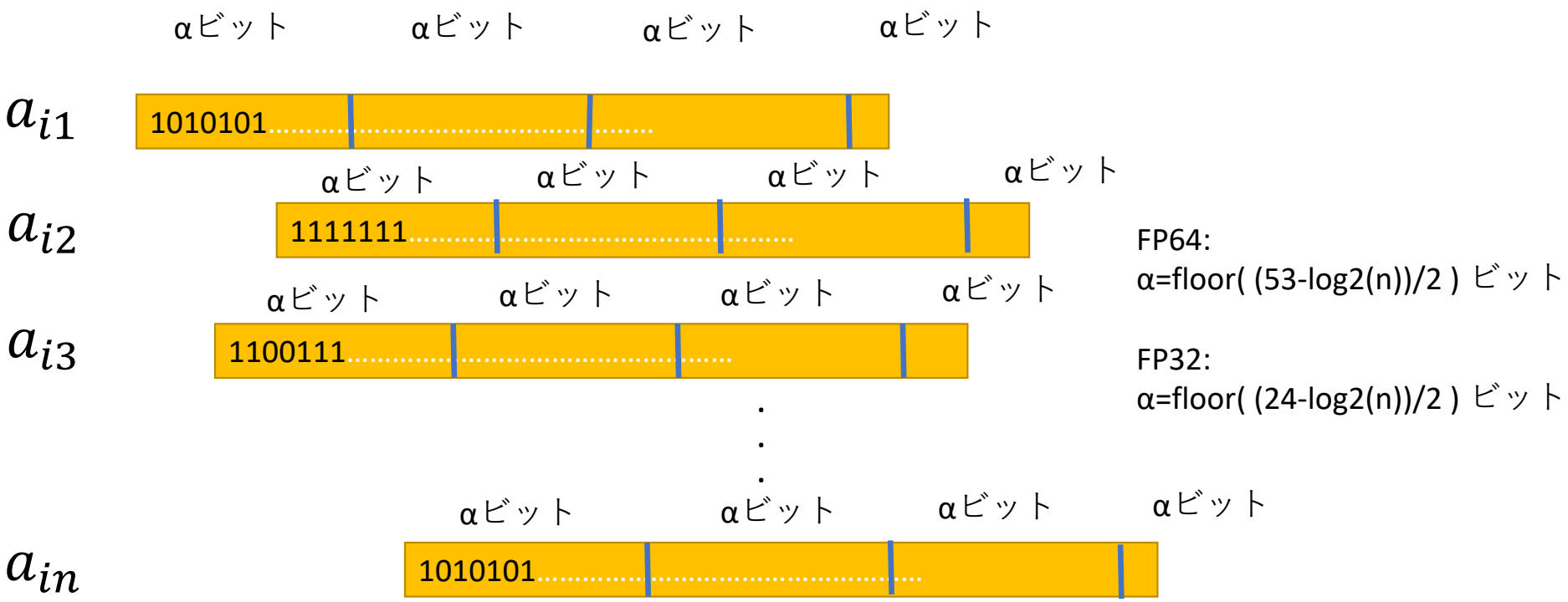
正しいイメージ



FP64:
α=floor((53-log2(n))/2)ビット

FP32:
α=floor((24-log2(n))/2) ビット

間違いなイメージ



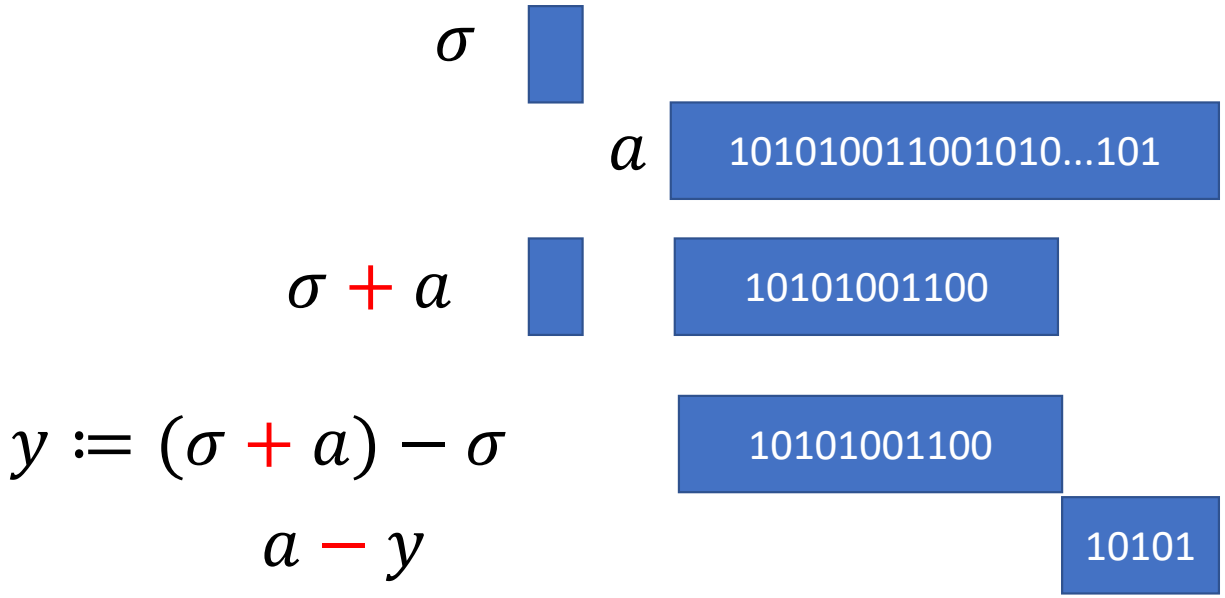
FP64:
α=floor((53-log2(n))/2) ビット

FP32:
α=floor((24-log2(n))/2) ビット

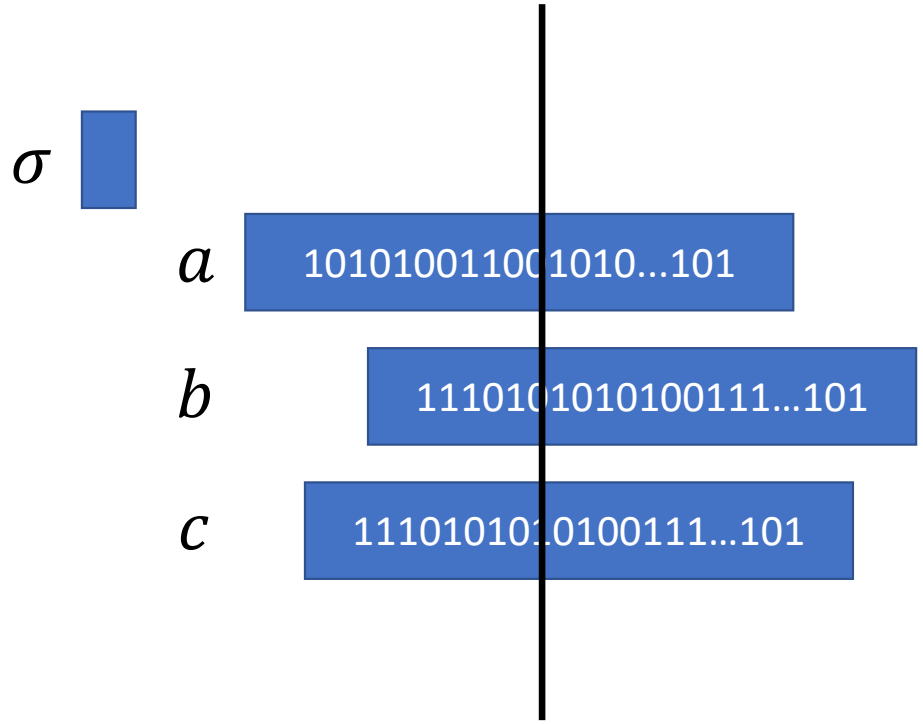
例：a*b+c*dみたいな計算が「大きい数×大きい数＋小さい数×小さい数」だったら？

FPに対する分割方法

- $y \coloneqq (\sigma + a) - \sigma,\ \sigma \gg a$
- 浮動小数点演算



FPに対するスライス方法



$(\sigma + a) - \sigma$

$(\sigma + b) - \sigma$

$(\sigma + c) - \sigma$

浮動小数点演算である！

整数ならばビットマスクも別の意味で使える

ビット数のイメージ (FP32)

- A がm行**n列**, B が**n行**p列
- $a_{ij}^{(k)}$ と $b_{ij}^{(k)}$ は最大で floor((24-log2(n))/2) ビットを持つ.

- n = 1,000 (≦2^{10})の例：
- $a_{ik}^{(*)}$ と $b_{kj}^{(*)}$ は最大で **7** ビット.
- $a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 14 ビット.
- $\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ 最大で 24 ビット (≦**24 ビット**).

ビット数のイメージ (FP32)

- A がm行**n列**, B が**n行**p列
- $a_{ij}^{(k)}$ と $b_{ij}^{(k)}$ は最大で floor((24-log2(n))/2) ビットを持つ

- n = 20,000 (≦2^{15})の例：
- $a_{ik}^{(*)}$ と $b_{kj}^{(*)}$ は最大で 4ビット.
- $a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 8ビット.
- $\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ は最大で 23 bit (≦**24ビット**).

ビット数のイメージ (FP64)

- A がm行**n列**, B が**n行**p列
- $a_{ij}^{(k)}$ と $b_{ij}^{(k)}$ は最大で floor((53-log2(n))/2) ビットを持つ

- n = 1,000 (≦2^{10})の例：
- $a_{ik}^{(*)}$ と $b_{kj}^{(*)}$ は 最大で21 ビット.
- $a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 42 ビット.
- $\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ は最大で 52 ビット(**≦53 ビット**).

ビット数のイメージ (FP64)

- A がm行**n列**, B が**n行**p列
- $a_{ij}^{(k)}$ と $b_{ij}^{(k)}$ は最大で floor((53-log2(n))/2) ビットを持つ

- n = 10,000 (≦2^{14})の例：
- $a_{ik}^{(*)}$ は $b_{kj}^{(*)}$ は最大で 19 ビット.
- $a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 38ビット.
- $\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ は最大で 52 ビット (**≦53 ビット**).

ちなみにAとBの要素が持つビット数のバランスも変えられます．そういう論文もあります

スライスとスプリット

- １０進法の数値例

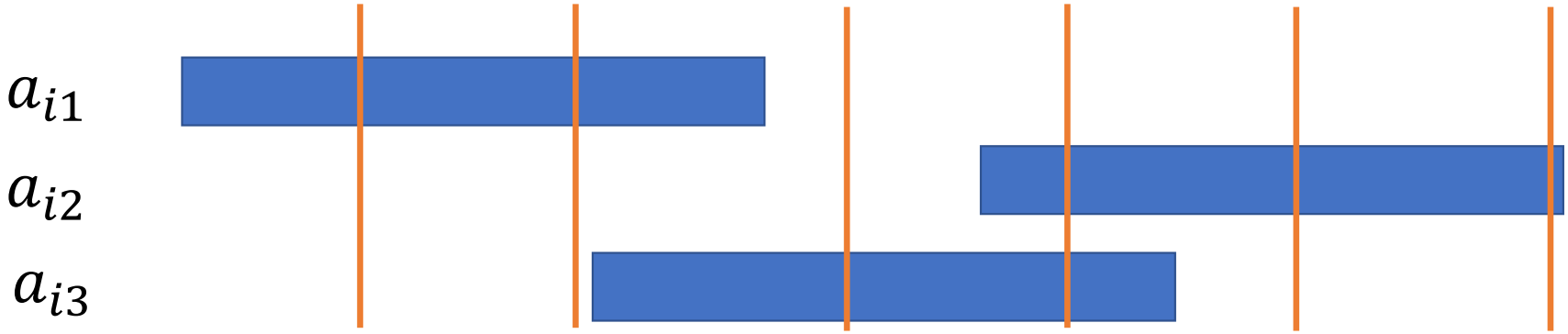
- 99999999を99000000, 990000, 9900, 99に分ける（スライス）
（２進法でいうビットマスク的な分割方法）

- 99999999を100000000, -1に分ける
（足して引くスプリット方式）

短所

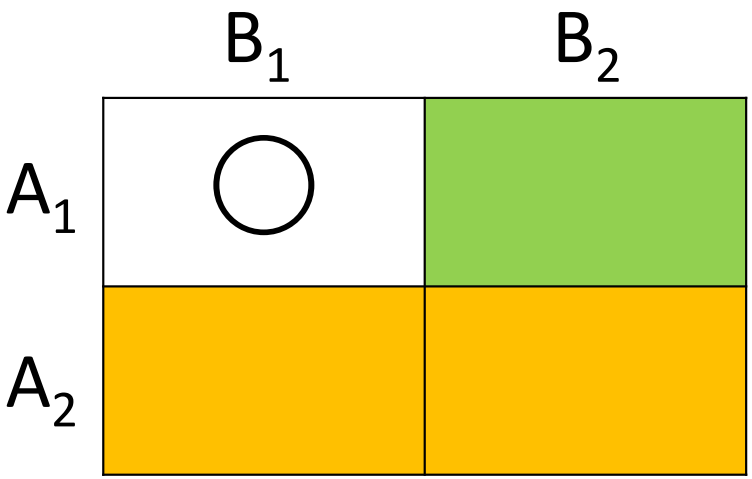
尾崎スキームIはいつでも有用ではない！

- 行列を分割するのでメモリをたくさん使う
- 成分間の絶対値の差が大きいとき？
- サイズによって，保持するビット数が変わる

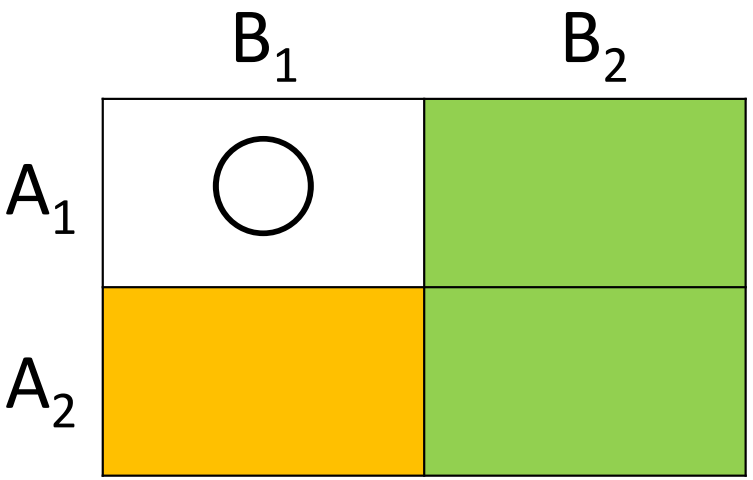


○は無誤差で計算

２スライス版



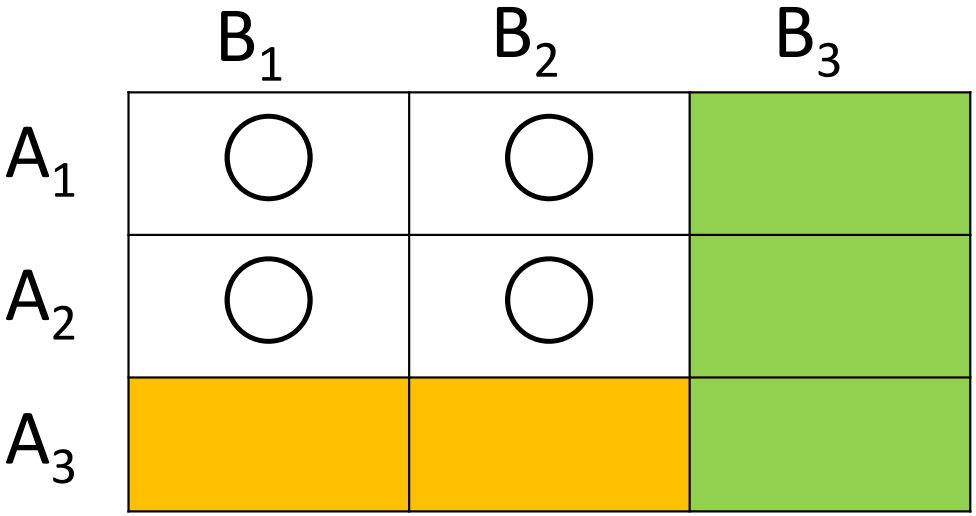
$A_1B_1+A_1B_2+A_2B$



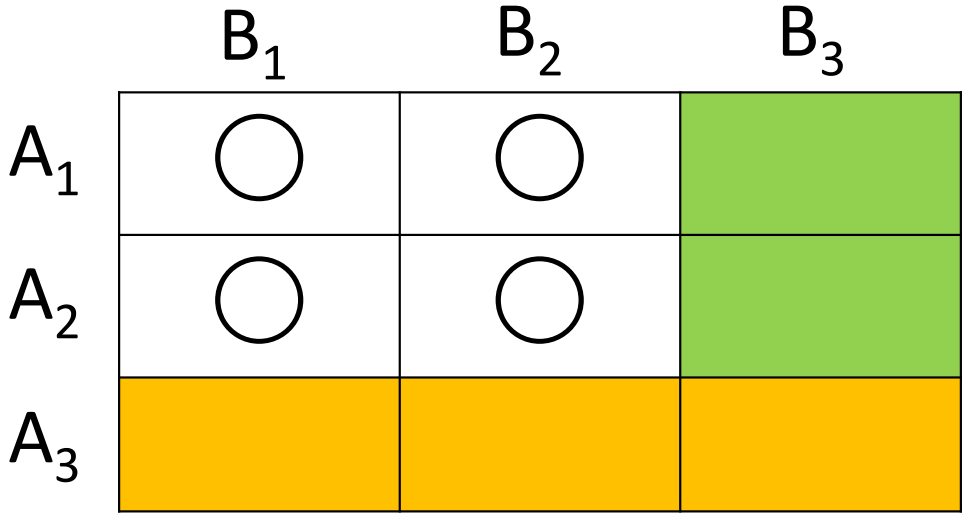
$A_1B_1+A_2B_1+AB_2$

○は無誤差で計算

3スライス版



$A_1B_1+A_1B_2+A_2B_1+A_2B_2+A_3(B_1+B_2)+AB_3$



$A_1B_1+A_1B_2+A_2B_1+A_2B_2+(A_1+A_2)B_3+A_3B$

最も単純なコード（MATLABとJulia）

```
function C = Mul3(A,B)
n = size(A,2);
beta = ceil((53 + log2(n))/2);
%% Split A
[mi,ma] = bounds(A,2);
mu = max(ma,-mi);
w = 0.75*2.^(ceil(log2(mu)) + beta);
A1 = (A + w) - w;
A2 = A - A1;
%% Split B [mi,ma] = bounds(B,1);
mu = max(ma,-mi);
w = 0.75*2.^(ceil(log2(mu)) + beta);
B1 = (B + w) - w;
B2 = B - B1;
%% Compute matrix products and the sum
C = A1 * B1 + (A1 * B2 + A2 * B);
end
```

```
function Mul3(A::Matrix{Float64}, B::Matrix{Float64})
    n = size(A, 2)
    beta = ceil(Int, (53 + log2(n)) / 2)
    # ==== Split A (行方向：dims=2) ====
    muA = vec(maximum(abs, A; dims=2))
    wA  = reshape(0.75 .* 2.0 .^ (ceil.(log2.(muA)) .+ beta), :, 1)
    A1 = (A .+ wA) .- wA
    A2 = A .- A1
    # ==== Split B (列方向：dims=1) ====
    muB = vec(maximum(abs, B; dims=1))
    wB  = reshape(0.75 .* 2.0 .^ (ceil.(log2.(muB)) .+ beta), 1, :)
    B1 = (B .+ wB) .- wB
    B2 = B .- B1
    # compute matrix products and the sum
    return A1 * B1 + (A1 * B2 + A2 * B)
end
```

A1*B1+A1*B2+A2*B

数値実験（入力FP64・内部DD・出力FP64）

n=2048, 最大相対誤差（行列積は悪条件）

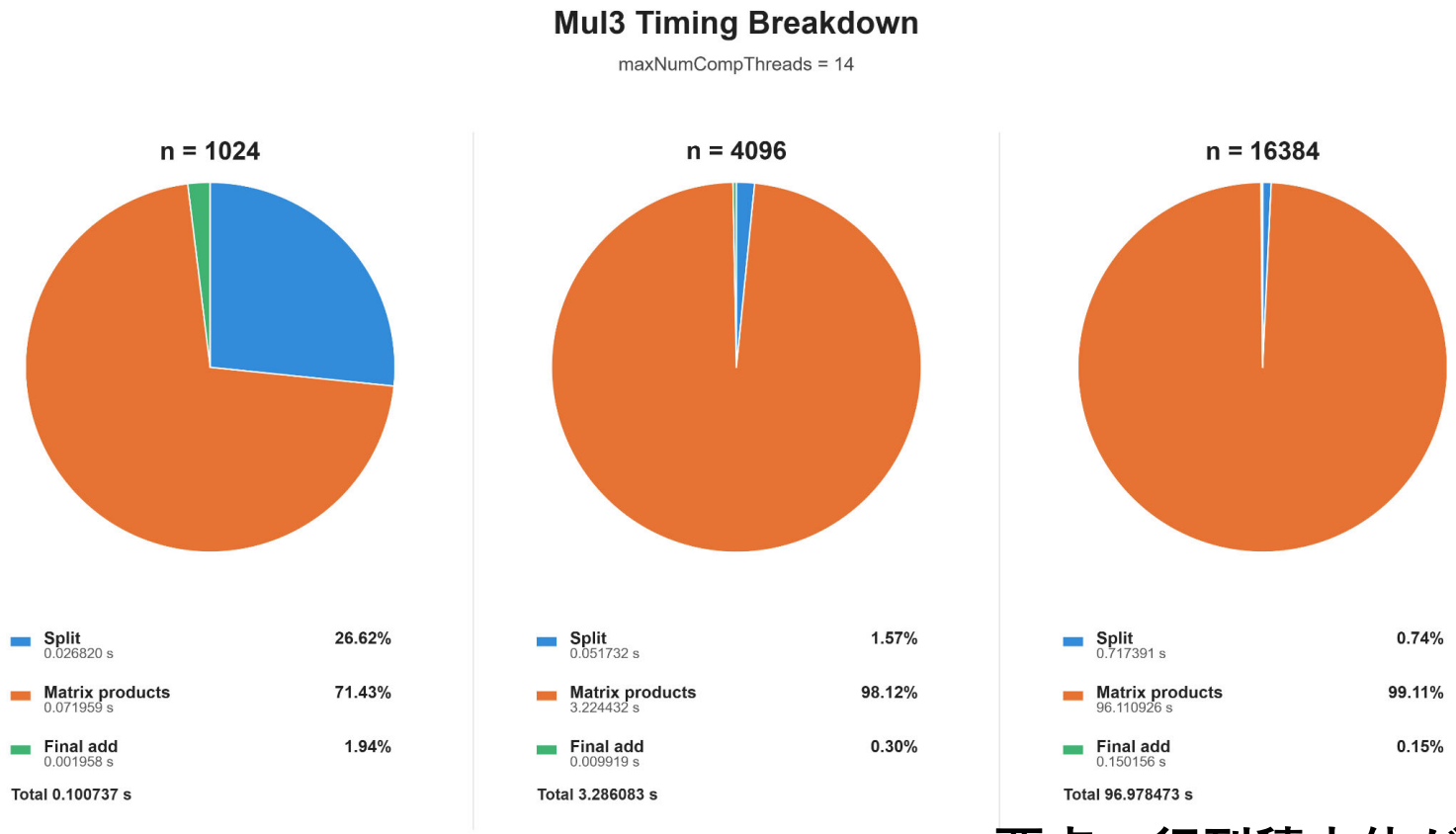
積の回数	1 (FP64)	3	4	5	6	8	9	10
Ozaki Scheme	4.34e+05	9.76e-02	3.62e-04	3.78e-04	1.74e-07	4.57e-10	**2.37e-12**	8.48e-14
Julia MF2	**6.94e-12**							
Advanpix	1.11e-16							

n=2048, 計算時間（秒）

積の回数	1 (FP64)	3	4	5	6	8	9	10
Ozaki Scheme	M：0.06 J：0.08	M：0.21 J：0.39	M：0.31 J：0.49	M：0.37 J：0.58	M：0.45 J：0.64	M：0.57 J：0.85	**M：0.66** **J：0.94**	M：0.71 J：1.05
Julia MF2	**7.82**							
Advanpix	**40.0**							

MF2はMultiFloatsでFloat64x2の意味（自分で並列化）
AdvanpixはMPFRベースの高速な実装（有料ソフト）

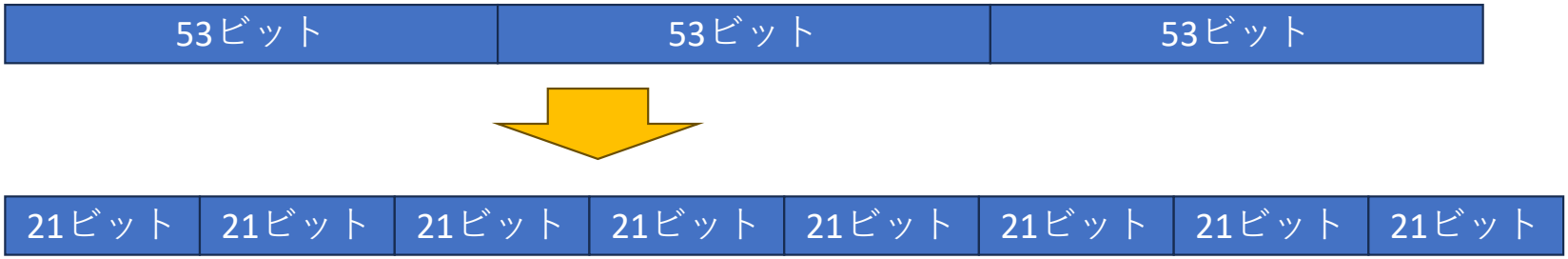
どこに時間がかかるのか？



要点：行列積本体が支配的

マルチワード演算にも応用可能

- Triple-wordの例（n=1024）



- 各行列積はFP64のDGEMMを使用して計算
- 行列積計算後の加算はTriple-word arithmeticを使用

尾崎スキームI for GPU

ー２０２０年ちょっと前からはじまった研究ー

低精度演算「超」高速時代

TFLOPS / TOPS for data center class GPUs

GPU	INT8 TC	FP16 TC	FP32	FP64 TC	FP64
B200	4500	2250	75	37	37
GH200	1979	989	67	67	34
A100	624	312	19.5	19.5	9.7

new

TFLOPS / TOPS for consumer grade GPUs

GPU	INT8 TC	FP16 TC	FP32	FP64 TC	FP64
RTX5090	1674	419	104.8	—	1.64
RTX4090	660	165	82.6	—	1.29

new

INT8TC:FP64 = 121:1

INT8TC:FP64 = 1020:1

INT8TC:FP64 = 30:1

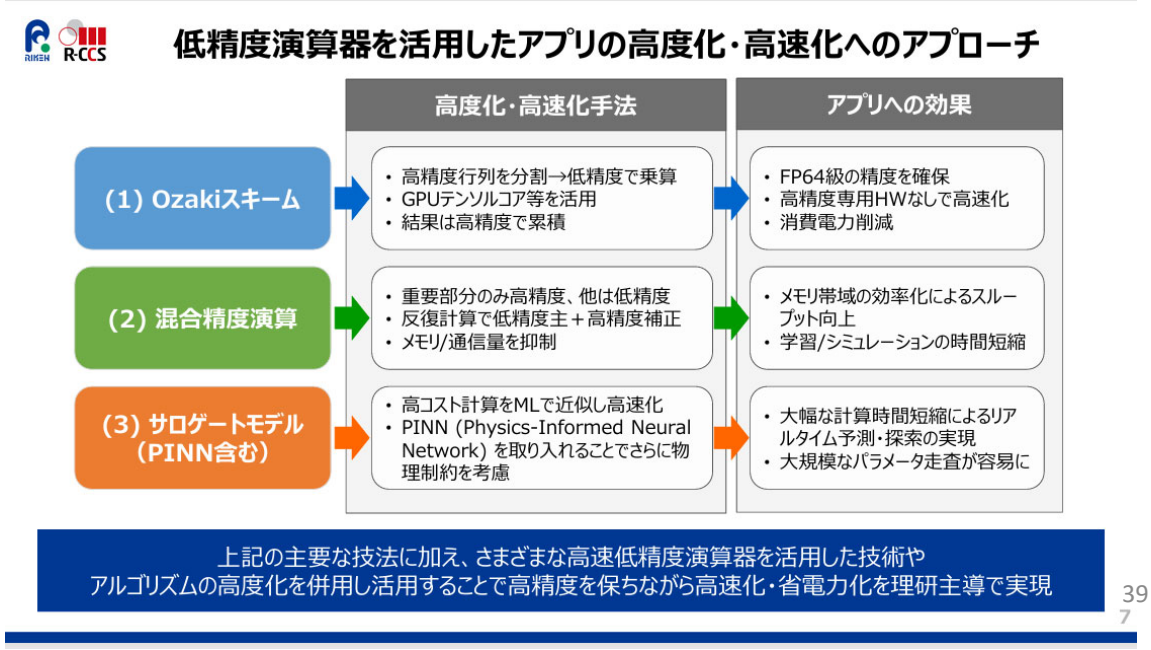
INT8TC:FP64 = 512:1

FP64 Emulation

GPU関連の尾崎スキーム

- 2025年8月22日 HPCI計画推進委員会
- 「理化学研究所を中核とした国際連携による「富岳NEXT」開発体制とその意義」から抜粋

https://www.mext.go.jp/content/20250822-mxt-jyohoka01-000044312_41.pdf



GPU関連の尾崎スキーム

- CUDA Toolkit 13.0 Update 2　（2025年10月）

2.3. New Features

CUDA 13.0 is a new major version. CUDA follows semantic versioning, and guarantees ABI stability within the major version series. CUDA Toolkit releases in the 13.x series are ABI-compatible with drivers corresponding to the same series (r580 and newer). Note that CUDA's API can evolve over the course of a major release to include new functionality, and not all new functionality may be supported on older drivers.

2.3.1. General CUDA

- Enabled opt-in fixed-point emulation for FP64 matmuls (D/ZGEMM) which improves performance and power-efficiency. The implementation follows the Ozaki-1 Scheme and leverages an automatic dynamic precision framework to ensure FP64-level accuracy. See here for more details on fixed-point emulation along with the table of supported compute-capabilities and the CUDA library samples for example usages.

参照：
https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html

NVIDIA Technical Blog

- **Unlocking Tensor Core Performance with Floating Point Emulation in cuBLAS**

- https://developer.nvidia.com/blog/unlocking-tensor-core-performance-with-floating-point-emulation-in-cublas/

- FP32・FP64エミュレーションについて述べている.

※ 正規化数の暗黙の1ビットを含む

テンソルコア

- FP8 TC
  - 入力: FP8 (仮数部4 or 3ビット), 出力: FP16 (仮数部11ビット)
  - 入力: FP8 (仮数部4 or 3ビット), 出力: FP32 (仮数部24ビット)
- FP16 TC
  - 入力: FP16 (仮数部11ビット), 出力: FP16 (仮数部11ビット)
  - 入力: FP16 (仮数部11ビット), 出力: FP32 (仮数部24ビット)
- BF16 TC
  - 入力: BF16 (仮数部8ビット), 出力: FP32 (仮数部24ビット)
- INT8 TC
  - input : INT8 (仮数部7ビットみたいなもの)
    output: INT32 (仮数部31ビットみたなもの)

FP16 Tensor Cores

n = 1,000,

$a_{ik}^{(*)}$ と $b_{kj}^{(*)}$ は最大で **7 ビット**.

$a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 14 ビット.

$\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ は最大で **24 ビット**.

FP16で十分
スケーリング有

FP32に収まる

INT8 Tensor Cores

- 入力: 8 ビット 整数 (仮数部的なイメージで 7 ビット)

- 出力: **32 ビット 整数** (仮数部的に 31 ビット)

$a_{ik}^{(*)}$ と $b_{kj}^{(*)}$ は最大で **7** ビット.

$a_{ik}^{(*)} * b_{kj}^{(*)}$ は最大で 14 ビット.

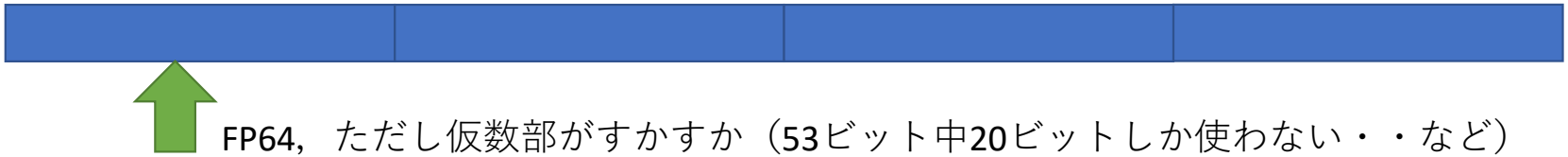
$\sum_{k=1}^{n} a_{ik}^{(*)} b_{kj}^{(*)}$ は最大で $14+\log_2 n$ ビット (≦31 ビット).

$n = 2^{17} \rightarrow 31$ビット

もちろんスケーリングは必須です

CPU・GPU版のメモリ使用について（FP64）

- A＝A1＋A2＋A3＋A4



FP64，ただし仮数部がすかすか（53ビット中20ビットしか使わない・・など）

- A＝A1＋A2＋A3＋A4＋A5＋A6＋A7+A8

INT8，無駄がない（スケーリング情報は別にもっている）

入力が低精度で良いスキームとテンソルコアの相性は抜群

INT8 TC v.s. FP16 Tensor Cores

最大相対誤差（n=5000,　疑似乱数行列）

スライス数	Muls	FP16 TC	INT8 TC
FP32	[redacted]	1.1027e+00	
2	3	3.1601e+05	5.1525e+03
3	6	9.1655e+03	5.5134e+01
4	10	6.6406e+01	4.3142e-01
5	15	1.7326e+00	3.0162e-03
6	21	1.9342e-02	1.9846e-05
7	28	6.6692e-04	1.2183e-07
8	36	8.1598e-06	6.5806e-10
9	45	9.6758e-08	3.6994e-12
FP64	[redacted]	3.4942e-09	

精度のコントロールがスライス数によって可能

For B300, https://www.nvidia.com/ja-jp/data-center/hgx/

CUDA Toolkit 13.0 Update 2

- CUDAでdgemmと書いていた部分が高速になる.

export CUBLAS_EMULATE_DOUBLE_PRECISION=1
export CUBLAS_EMULATION_STRATEGY=performant

eager: 利用可能な場合はOzaki Scheme を使用
performant: 速くなる場合にのみOzaki Schemeを使用
自動調整を行う

- ビット数をcublasSetFixedPointEmulationMaxMantissaBitCount で指定する.

スライス数→ $\left\lceil \frac{bit数 + 1}{8} \right\rceil$

※ 固定ビット数指定ではFP64 と同等以上の精度が常に保証されるとは限らない

Level 3 BLAS ルーチン

- 倍精度（実）ルーチン

- DGEMM：一般の実行列同士の行列積
- DSYMM：対称行列と一般行列の積
- DSYRK ：行列とその転置の積による対称行列のランク更新
- DSYR2K：2 つの行列を用いた対称行列のランク 2k 更新
- DTRMM：三角行列と一般行列の積
- DTRSM ：三角行列を係数とする行列方程式

Level 3 BLAS ルーチンの比較

GFLOPS: CUBLAS_EMULATE_DOUBLE_PRECISION = **0**

サイズ	DGEMM	DSYMM	DSYRK	DSYR2K	DTRMM	DTRSM
1024	340	282	317	385	196	245
2048	317	280	301	363	280	252
4096	329	311	311	365	307	282
8192	329	322	323	394	342	326

GFLOPS: CUBLAS_EMULATE_DOUBLE_ PRECISION = **1**

サイズ	DGEMM	DSYMM	DSYRK	DSYR2K	DTRMM	DTRSM
1024	910	320	318	385	316	246
2048	1806	326	323	363	324	480
4096	2446	311	311	365	307	820
8192	2787	327	931	1083	348	1388

RTX5060 Tiを使用した実験結果です

Julia+GPU+Ozaki-I Scheme

- using CUDA
- n = 4096;
- A = CUDA.rand(Float64, n, n);
- B = CUDA.rand(Float64, n, n);
- C = A*B;

0.554 秒→231 GFLOPS

export CUBLAS_EMULATE_DOUBLE_PRECISION=1
export CUBLAS_EMULATION_STRATEGY=performant

- using CUDA ← V13.0.2以降
- n = 4096;
- A = CUDA.rand(Float64, n, n);
- B = CUDA.rand(Float64, n, n);
- C = A*B;

0.0745秒→**1718 GFLOPS**

MATLABは今はまだ無理？R2026a (prerelease)のToolkitのバージョンは12.8だった.

論文（CPU版）

- （**原点**）K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, “Error-Free Transformations of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and Its Applications,” Numerical Algorithms, Vol. 59, No. 1, pp. 95–118, 2012. DOI: 10.1007/s11075-011-9478-1.
- （**一般化**） K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, “Generalization of Error-Free Transformation for Matrix Multiplication and Its Application,” Nonlinear Theory and Its Applications, IEICE, Vol. 4, No. 1, pp. 2–11, 2013. DOI: 10.1587/nolta.4.2.
- （**事後保証型**） K. Ozaki, T. Ogita, and S. Oishi, “Error-Free Transformation of Matrix Multiplication with A Posteriori Validation,” Numerical Linear Algebra with Applications, Vol. 23, No. 5, pp. 931–946, 2016. DOI: 10.1002/nla.2061.
- （**非対称効率化**） K. Ozaki, D. Mukunoki, and T. Ogita, “Extension of Accurate Numerical Algorithms for Matrix Multiplication Based on Error-Free Transformation,” Japan Journal of Industrial and Applied Mathematics, Vol. 42, pp. 1–20, 2025. DOI: 10.1007/s13160-024-00677-z.

論文（GPU版）

- （**FP16TC使用**）D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, “DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions,” Lecture Notes in Computer Science, Vol. 12151, pp. 230–248, 2020. DOI: 10.1007/978-3-030-50743-5_12.
- （**INT8TC使用**） H. Ootomo, K. Ozaki, and R. Yokota, “DGEMM on Integer Matrix Multiplication Unit,” The International Journal of High Performance Computing Applications, Vol. 38, No. 4, pp. 297–313, 2024. DOI: 10.1177/10943420241239588.
- （**INT8TC使用・高速化**） Y. Uchino, K. Ozaki, and T. Imamura, “Performance Enhancement of the Ozaki Scheme on Integer Matrix Multiplication Unit,” The International Journal of High Performance Computing Applications, Vol. 39, No. 3, pp. 462–476, 2025. DOI: 10.1177/10943420241313064.

論文（GPU版）

- （**FP8TC使用**）D. Mukunoki, “DGEMM Using FP64 Arithmetic Emulation and FP8 Tensor Cores with Ozaki Scheme,” Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops, SCA/HPCAsiaWS ’26, Association for Computing Machinery, New York, NY, USA, pp. 303–311, 2026.
  DOI: 10.1145/3784828.3785017.
- （**１ビットボーナス**） A. Schwarz, A. Anders, C. Brower, H. Bayraktar, J. Gunnels, K. Clark, R. G. Xu, S. Rodriguez, S. Cayrols, P. Tabaszewski, and V. Podlozhnyuk, “Guaranteed DGEMM Accuracy While Using Reduced Precision Tensor Cores Through Extensions of the Ozaki Scheme,” Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region, SCA/HPCAsia ’26, Association for Computing Machinery, New York, NY, USA, pp. 91–101, 2026.
  DOI: 10.1145/3773656.3773670.

丸め誤差解析（GPU版）

- Y. Uchino, K. Ozaki, and T. Imamura,
  “Performance Enhancement of the Ozaki Scheme on Integer Matrix Multiplication Unit,”
  The International Journal of High Performance Computing Applications, Vol. 39, No. 3, pp. 462–476, 2025.
  DOI: 10.1177/10943420241313064.

- A. Abdelfattah, J. Dongarra, M. Fasi, M. Mikaitis, and F. Tisseur,
  “Analysis of Floating-Point Matrix Multiplication Computed via Integer Arithmetic,”
  arXiv:2506.11277v3 [math.NA], 2026.
  DOI: 10.48550/arXiv.2506.11277.

ライブラリ（GPU版）

- Daichi Mukunoki, ozblas, GitHub repository, https://github.com/mukunoki/ozblas.
- Hiroyuki Ootomo, ozIMMU, GitHub repository, https://github.com/enp1s0/ozIMMU.
- RIKEN-RCCS, accelerator_for_ozIMMU, GitHub repository, https://github.com/RIKEN-RCCS/accelerator_for_ozIMMU.
- NVIDIA, CUDALibrarySamples: cuBLASDx dgemm_emulation sample, GitHub repository, https://github.com/NVIDIA/CUDALibrarySamples/tree/master/MathDx/cuBLASDx/16_dgemm_emulation.

CPU版（さらに拡張されたもの）はINTLABで提供されている

まとめ

- 尾崎スキームI
- CPUでは高精度計算に有用
- GPUではFP64エミュレーションに有用

- メリット・デメリット
  - 精度のコントロールが可能
  - 場合によっては高速
  - メモリが必要
  - 成分間の絶対値の差があるときに非効率？

次回の内容（2026/07/23）

- 尾崎スキームIIの解説
  - 中国式剰余定理をベースにした近似計算
- 尾崎スキームの特徴
  - 数値再現性
  - 精度の細かなコントロール
- 尾崎スキームの応用
  - 数値線形代数への応用
  - アムダールの法則に縛られるか？
- 尾崎スキームの今後？