Vol.2021-HPC-180 No.7
2021/7/20

TensorCoreを用いた精度補正単精度行列積

大友 広幸[,a)]　横田 理央[,b)]

概要：NVIDIA TensorCore は最大 300TFlop/s 以上の性能を持つ混合精度行列積演算回路である．TensorCore は深層学習からの高い行列積需要に対応するために開発されたが，線型方程式の反復解法やフーリエ変換など，深層学習以外の分野への応用も研究されている．密行列積計算も深層学習に限らず幅広い分野において重要な計算である．TensorCore は入力として半精度（FP16）行列をとるため，これを用いて単精度（FP32）密行列積計算を行う場合は，はじめに入力行列を半精度へ変換する必要がある．しかしこの操作によって単精度度行列積の計算精度が劣化する．そこで入力行列を半精度へ変換する際に失われる仮数部を別の FP16 変数で保持し，これを用いて単精度行列積の計算精度を補正する手法が考案された．この手法では単精度演算器を用いた行列積と比較して高速に計算可能ではあるが，誤差の蓄積が大きく計算精度が悪いという問題が確認されている．本研究ではこの誤差蓄積の原因となる 2 つの問題に着目し，それらの改善を行うことで，単精度演算器で計算した場合と同等の計算精度でより高速な単精度行列積手法を開発した．この手法をオープンソースの行列積ライブラリである NVIDIA CUTLASS に実装し，様々な入力行列での計算精度・計算性能の評価を行った．計算性能では 40TFlop/s 以上の性能を実現した．

キーワード：TensorCore，精度補正，単精度行列積，混合精度，丸め

1. はじめに

深層学習の発展に伴い，Google TPU (Tensor Processing Unit)[12] や IBM POWER10[3]，ARMv8.6-a[13] など，行列計算に特化した回路を持つプロセッサが多く開発されている．NVIDIA TensorCore [14] は混合精度行列積和演算回路で，現在その理論性能は最大で 312 TFlop/s である [4]．TensorCore では入力行列を半精度（FP16; IEEE 754 Binary16），内部の演算を完全精度・単精度（FP32; IEEE 754 Binary32）で行うことで，FP16 演算器を用いた場合と比較して高い計算精度を実現している．この TensorCore を深層学習以外の科学技術計算に用いる研究が行われている．線型方程式の反復解法の一部に TensorCore による混合精度演算を用い，高速に線型方程式の数値解を得ることが可能となった [10]．TensorCore は半精度・単精度を取り扱う演算器であるが，この手法は倍精度問題に対しても利用可能な手法であり，現在では cuSOLVER の実装としても用いられている[*1]．Graph analysis や幅優先探索，Multigrid 法に用いられる疎行列積を TensorCore を用いることで高速に計算する研究 [18] や，reduction/scan を TensorCore を用いて行う研究 [2], [6], [9] が行われている．

一方で，TensorCore の演算器の調査やその誤差解析も行われている．演算器の調査では，TensorCore の命令である HMMA が入力行列をどう部分行列へ分割し，どの順番でそれらの積和を計算を行っているか [11], [17] や，TensorCore 内部の Accumulator の長さ，丸めについての調査が行われている [7]．誤差解析では，TensorCore を用いた場合の丸め誤差の限界の調査が行われている [1].

TensorCore は入力として FP16 を取るため，TC で単精度行列積を計算する場合は FP16 へ変換して入力する必要がある．このため FP32 SIMT Core による単精度行列積と比較して計算精度が劣化する．これに対し，TensorCore の内部での計算が高精度で行われる特長を活用し，TensorCore を複数回用いて行列積を高精度で計算する手法が提案されている．椋木らは TensorCore を Ozaki scheme[16] に用いることで，単精度及び倍精度行列積の計算を可能とした [15]．この手法は FP64 演算器が遅い GeForce 等の一般消費者向けの GPU では cuBLAS と比較して高速に倍精度行列積を計算できる．しかし，単精度行列積や高速な FP64 演算が可能な NVIDIA Tesla シリーズでの倍精度行

[†1] 現在，東京工業大学
Presently with Tokyo Institute of Technology
a) ootomo.h@rio.gsic.titech.ac.jp
b) rioyokota@gsic.titech.ac.jp

[*1] `cusolverIRSRefinement_t` section of cuSOLVER Documentation https://docs.nvidia.com/cuda/cusolver/

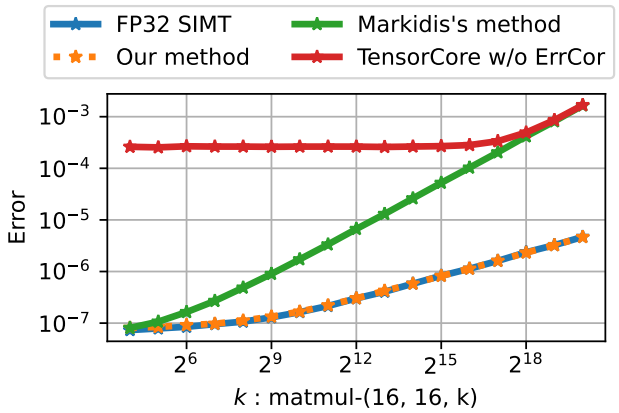

図 1: 本研究の精度補正手法，Markidis らの精度補正手法，FP32 SIMT Core により計算した単精度行列積の精度比較

列積演算は，cuBLAS と比較して遅いという課題がある．単精度行列に限れば，Markidis らは TensorCore での行列積に精度補正計算を加え，単精度行列積に近い精度での計算を行う手法を提案した [14]．しかしこの研究では精度の補正が十分ではなく，FP32 SIMT Core での単精度行列積と比較して計算精度が悪いという問題がある．これに対し Feng らは，精度補正計算中により多くの仮数部長を保持できるよう改善を行ったと主張しているが，精度の評価手法に問題があり，この主張には疑問が残る．

本研究では Markidis らの手法に注目し，この計算精度の向上及び計算性能の向上を行った．Markidis の手法 [14] には 2 つの問題点がある．1 つ目は TensorCore 内での演算では，丸めとして RZ 丸めが用いられており [7]，誤差の蓄積が大きい問題である．2 つ目は精度補正のための入力行列の各要素がアンダーフローもしくは漸進アンダーフローを起こす確率が高い問題である．これらの問題を解決した本研究での精度補正単精度行列積の精度を，FP32 SIMT Core による行列積，Markidis らの精度補正単精度行列積と比較したものが図 1 である．さらに本研究は計算精度を改善した上で，Markidis の手法や Feng らの手法 [8] にある無駄な計算を排除することで計算量の削減を行った．これを NVIDIA CUTLASS へ移植し，計算精度及び計算性能の評価を行った．

表記

toF16(v)

単精度変数 v を半精度へ変換する．断りのない限り丸めには RN 丸めを用いる．

toF32(v)

半精度変数 v を単精度へ変換する．

matmul-(m, n, k)

$m \times k$ 行列 $\mathbf{A}$，$k \times n$ 行列 $\mathbf{B}$，$m \times n$ 行列 $\mathbf{C}$ に対する行列積．

2. 背景

2.1 Tensor Cores

NVIDIA Ampere アーキテクチャに搭載される第三世代 TensorCore では最大 300TFlop/s 以上の計算性能を持つ．FP16 の仮数部長は $10+1$ bit であり，FP32 の仮数部長 $23+1$ bit と比べて小さく計算精度が悪い．このため，FP32 行列積を TensorCore を用いて計算する場合は入力行列を FP16 へ変換することが必要であり，最終的な計算精度は FP32 SIMT Core を用いて計算した場合と比較して劣化する．しかし TensorCore では，行列積内部の積算を高精度（完全精度; Full precision）で行うことで，単に FP16 SIMT Core を用いて演算をした場合と比較して計算精度の劣化を抑えられる．

CUDA では TensorCore を用いるための API である WMMA API を提供している．WMMA API では fragment と呼ばれるレジスタ配列を用いてレジスタブロッキングを行う．WMMA API を用いた行列積計算 $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$ の疑似コードをソースコード 1 に示す．この例では，行列 $\mathbf{A}, \mathbf{B}$ をそれぞれ 16×16 のいくつかの部分行列に分割し，それらの積和を行うことで計算を完了する．`load_matrix_sync`，`mma_sync`，`store_matrix_sync` 関数は WMMA API の関数である．

```
1  void matmul(mem_c, mem_a, mem_b) {
2    fragment<16, 16, 16> frag_a, frag_b, frag_c;
3    // 1; accumulator fragmentの初期化
4    fill_fragment(frag_c, 0.f);
5    for (k = 0; k < K; k += 16) {
6      // 2; A, Bの部分行列のfragmentへの読み込み
7      load_matrix_sync(frag_a, mem_a + k);
8      load_matrix_sync(frag_b, mem_b + k);
9      // 3; TensorCoreによる部分行列の積和計算
10     mma_sync(frag_c, frag_a, frag_b, frag_c);
11   }
12   // 4; メモリへの書き出し
13   store_matrix_sync(mem_c, frag_c);
14 }
```

ソースコード 1: TensorCore を用いた行列積計算の疑似コード

2.2 精度補正単精度行列積

FP32 行列積を TensorCore を用いて計算する場合は精度が劣化するため，Markidis らはこれを補正する手法を考案した [14]．この手法では，FP32 行列を FP16 へ変換する際に失われる仮数部を別途保持しておき，これを用いて精度の補正を行う．今，FP32 行列 $\mathbf{A}_{\mathrm{F32}}, \mathbf{B}_{\mathrm{F32}} \in \mathrm{F32}^{n \times n}$ の積 $\mathbf{C}_{\mathrm{F32}} = \mathbf{A}_{\mathrm{F32}} \times \mathbf{B}_{\mathrm{F32}}$ を計算する．このとき Markidis らの手法では式 (1-5) により $\mathbf{C}_{\mathrm{F32}} \sim \hat{\mathbf{C}}_{\mathrm{F32}} = \mathbf{A}_{\mathrm{F32}} \times \mathbf{B}_{\mathrm{F32}}$ を計算する．

$$\mathbf{A}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\mathbf{A}_{\mathrm{F32}}\right) \tag{1}$$

$$\Delta\mathbf{A}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\mathbf{A}_{\mathrm{F32}} - \mathrm{toF32}\left(\mathbf{A}_{\mathrm{F16}}\right)\right) \tag{2}$$

$$\mathbf{B}_{\mathbf{FP16}} \leftarrow \mathrm{toF16}\left(\mathbf{B}_{\mathrm{F32}}\right) \tag{3}$$

$$\Delta\mathbf{B}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\mathbf{B}_{\mathrm{F32}} - \mathrm{toF32}\left(\mathbf{B}_{\mathrm{F16}}\right)\right) \tag{4}$$

$$\hat{\mathbf{C}}_{\mathrm{F32}} \leftarrow \mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} + \Delta\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} + \mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}} + \Delta\mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}} \tag{5}$$

式 1-5 を WMMA API を用いて実装した場合の疑似コードがソースコード 2 である．compute_delta 関数は自分で実装を行う．

```
1  void matmul(mem_c, mem_a, mem_b) {
2    fragment<16,16,16> frag_a, frag_b, frag_c;
3    fragment<16,16,16> frag_da, frag_db;
4    // accumulator fragmentの初期化
5    fill_fragment(frag_c,0.f);
6    for (k=0;k<K; k+=16) {
7      // A, Bの部分行列のfragmentへの読み込み
8      load_matrix_sync(frag_a,mem_a+k);
9      load_matrix_sync(frag_b,mem_b+k);
10     // A, Bの部分行列に対し式2,4を計算
11     compute_delta(mma_delta_a,mem_a + k);
12     compute_delta(mma_delta_b,mem_b + k);
13     // fragmentへの読み込み
14     load_matrix_sync(frag_da,mem_delta_a);
15     load_matrix_sync(frag_db,mem_delta_b);
16     // TensorCoreによる部分行列の積和計算
17     mma_sync(frag_c,frag_da,frag_db,frag_c);
18     mma_sync(frag_c,frag_da,frag_b,frag_c);
19     mma_sync(frag_c,frag_a,frag_sb,frag_c);
20     mma_sync(frag_c,frag_a,frag_b,frag_c);
21   }
22   // メモリへの書き出し
23   store_matrix_sync(mem_c,frag_c);
24 }
```

ソースコード 2: Markidis らの精度補正計算の疑似コード

Markidis ら手法で計算した単精度行列積の精度を図 1 に示す．誤差の評価には入力単精度行列 $\mathbf{A}_{\mathrm{FP32}}, \mathbf{B}_{\mathrm{FP32}}$ を倍精度へ変換し，cublasDGEMM を用いて計算した行列積 $\mathbf{C}_{\mathrm{FP64}} = \mathrm{F64}\left(\mathbf{A}_{\mathrm{FP32}}\right) \times : w\mathrm{F64}\left(\mathbf{B}_{\mathrm{FP32}}\right)$ をリファレンスとして用いる．それぞれの実装での行列積 $\mathbf{C}_{\mathrm{FP32}} = \mathbf{A}_{\mathrm{FP32}}\mathbf{B}_{\mathrm{FP32}}$ に対し，$\mathbf{C}_{\mathrm{FP64}}$ に対する相対誤差

$$\mathrm{Error} = \frac{||\mathbf{C}_{\mathrm{FP64}} - \mathbf{C}_{\mathrm{FP32}}||_F}{||\mathbf{C}_{\mathrm{FP64}}||_F} \tag{6}$$

を評価値として用いる（$||\cdot||_F$ はフロベニウスノルム）．Markidis らの手法では精度補正を行わない TensorCore 行列積と比較して精度が良いが，足し込みサイズが大きくなるに連れ FP32 SIMT Core を用いた場合と比較して精度が劣化する．Feng らは式 3,5 での仮数部の損失を少なくする手法を考案し実装を行ったが，リファレンス行列が倍精度ではなく単精度で計算されているため適切な評価ではなく，その精度は不明である [8].

3. 誤差の原因の解析及び改善

3.1 保持される仮数部長の期待値

式 1-2，3-4 で，任意の要素に対する計算は式 7-8 で表される．

$$v_{\mathrm{F16}} \leftarrow \mathrm{toFP16}\left(v_{\mathrm{F32}}\right) \tag{7}$$

$$\Delta v_{\mathrm{F16}} \leftarrow \mathrm{toFP16}\left(v_{\mathrm{F32}} - \mathrm{toFP32}\left(v_{\mathrm{F16}}\right)\right) \tag{8}$$

Feng らは式 8 での丸めに問題があり，これによって $v_{\mathrm{F16}} + \Delta v_{\mathrm{F16}}$ により保持される v_{F32} の仮数部長が小さくなるために，行列積の精度が悪くなるとした．しかし Feng らが計算した保持される仮数部長にはケチビットが考慮されていないなど多くの問題がある．そこで本稿では改めて式 7-8 により保持される仮数部長の期待値を 1 つの仮定のもとで計算する．

v_{F32} の仮数部を MSB (Most Significant Bit) から $m_{22}m_{21}\cdots m_0$ と表す．式 7，8 の toF16 では，NVIDIA GPU で実際に使われている RN 丸め (Round to nearest; ties to even) を用いるとする．このとき，$m_{13}\cdots m_0$ の bit により式 7 の toF16 の RN 丸めで繰り上がりが発生するかが決定される．この丸めの影響を考慮した上で，m_i の値によって場合分けをし，$v_{\mathrm{F16}} + \Delta v_{\mathrm{F16}}$ によって保持される仮数部長とその出現確率をまとめたものが表 1 である．表中の l_0 は m_{12} から LSB (Least Significant Bit) に向けて連続する 0 の個数である．この発現確率を計算するに当たり次の仮定 1 を用いた．

仮定 1 仮数部の各 bit が 0 か 1 となる確率は等確率で，その確率は bit 間で独立である

これより，仮数部長 23bit の v_{FP32} が $v_{\mathrm{F16}} + \Delta v_{\mathrm{F16}}$ によって保持される仮数部の期待値は 22.75bit である．この期待値は実験的にも確認されている．また，同様の議論により，丸めに RNA (Round to nearest; ties to away from zero) を用いた場合も保持される仮数部長の期待値は 22.75bit である．

この期待値 22.75bit の精度への影響を評価するため，より仮数部長の期待値が小さい 22.5bit での単精度行列積を行い，この精度と Markidis らの手法の精度を比較した．FP32 の仮数部 23bit のうち，LSB を強制的に 0 とした場合に保持される仮数部の期待値は，仮定 1 を用いて 22.5bit と計算される．単精度行列を計算する前処理として，仮数部の LSB を強制的に 0 とした場合の精度を評価したものが図 2 である．この結果，仮数部長期待値 22.75bit を表せる Markidis らの手法は仮数部長期待値 22.5bit の SGEMM よりも精度が悪く，Markidis らの手法の精度補正計算の精度が悪い原因は仮数部長の問題ではないことが分かる．

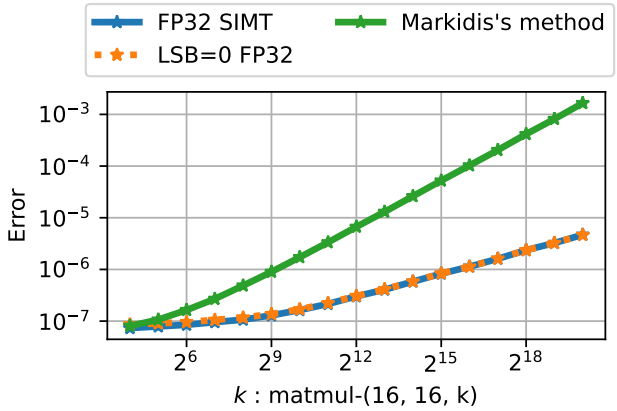

図 2: FP32 の仮数部長を上位 22bit でマスクした場合の matmul-$(16, 16, k)$ の精度評価．$\mathbf{A}, \mathbf{B}$ は (-8, 8) の一様乱数で初期化．

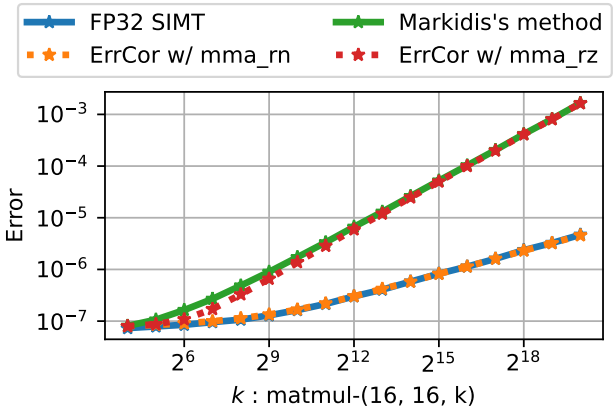

図 3: TensorCore を模した行列積演算関数を用いた精度補正計算による精度の評価．

l_0	m_{13}	m_{12}	m_{11}	m_1	m_0	len	prob
≥ 2	*	0	0	*	*	23	1/4
$=1$	*	0	1	*	0	23	1/8
	*	0	1	*	1	22	1/8
$=0$	*	1	0	*	1	22	1/8
	*	1	0	*	0	23	1/8
	*	1	1	*	*	23	1/4

表 1: v_{F32} の仮数部 23bit を $m_{22}m_{21}\cdots m_0$ と表した場合の v_{F16} と Δv_{F16} により保持される仮数部長 (len) とその発現確率 (prob)．表中の l_0 は m_{13} から LSB に向けて連続する 0 の個数 である．発現確率は仮定 1 を用いて計算．* は 0 か 1 かを問わない．

3.2 TensorCore を用いた場合の accumulation に RZ 丸めの回避

TensorCore の内部で行われる accumulation では，少なくとも 24bit の accumulator を用いており，丸めには RZ 丸めが用いられていることが確認されている [7]．すなわち，ソースコード 1 の k ループ内では毎ループ `frag_c` に対して RZ 丸めが行われる．この RZ 丸めの精度補正計算に与える影響を，TensorCore の内部計算を模した 2 種類の行列の積和関数，`mma_rn` 及び `mma_rz` を用いることで検証する．これらの関数は FP16 行列 $\mathbf{A}_{\mathrm{F16}}, \mathbf{B}_{\mathrm{F16}} \in \mathrm{FP16}^{16\times16}$, $\mathbf{C}_{\mathrm{F32}}, \in \mathrm{FP32}^{16\times16}$ を入力とし，これらの行列積和計算 $\mathbf{D}_{\mathrm{F32}} \leftarrow \mathbf{A}_{\mathrm{F16}} \times \mathbf{B}_{\mathrm{F16}} + \mathbf{C}_{\mathrm{F32}}$ を FP32 で行う．この 2 つの関数の違いは，`mma_rn` は $+\mathbf{C}_{\mathrm{F32}}$ の際に RN 丸めを用いるのに対し，`mma_rz` では RZ 丸めを用いる部分のみである．$\mathbf{A}_{\mathrm{F16}} \times \mathbf{B}_{\mathrm{F16}}$ 計算では TensorCore 同様 RZ 丸めを用いた．

この `mma_rn` 及び `mma_rz` を TensorCore の代わりに用い，精度補正計算により単精度行列積を計算した計算精度が図 3 である．`mma_rn` を用いた場合，その計算精度は FP32 SIMT と同程度なのに対し，`mma_rz` を用いた場合は Markidis らの手法と同程度であることが分かる．以上より，Markidis ら精度補正計算の精度劣化の原因は $+\mathbf{C}$ での RZ 丸めにあることが確認された．

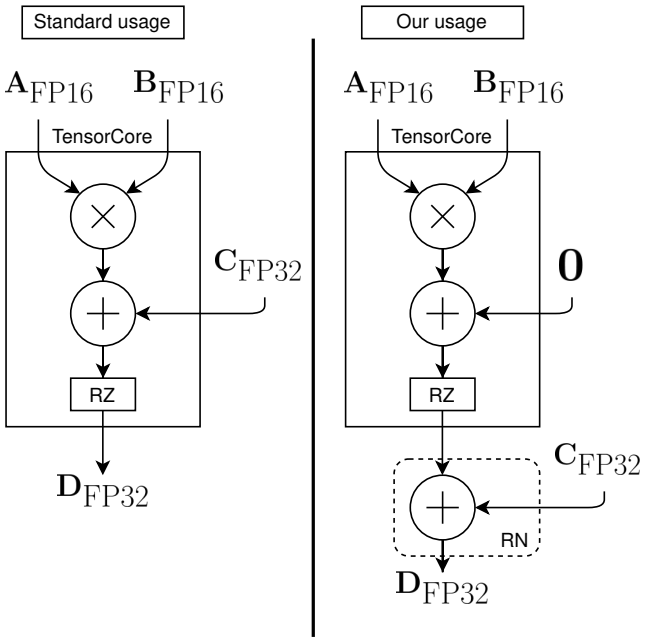

図 4: Accumucaltor C に対する直接的な TensorCore 内部での RZ 丸めを回避した行列積和計算．左は TensorCore を用いた標準的な行列積和計算の流れであり，Accumulator $\mathbf{C}_{\mathrm{FP32}}$ に対して RZ 丸めが発生する．右は本研究の提案する Accumulator $\mathbf{C}_{\mathrm{FP32}}$ に対する直接的な RZ 丸めを回避した TensorCore による行列積和の流れ．

そこで，この $+\mathbf{C}$ での RZ 丸めによる誤差蓄積の影響を緩和するよう Markidis らの手法を改良する．本研究では図 4 に示すように，$+\mathbf{C}_{\mathrm{F32}}$ の計算を TensorCore で行わず，FP32 SIMT Core で行う．これにより $\mathbf{C}_{\mathrm{F32}}$ に対する直接的な RZ 丸めは行われず，精度の改善が期待される．これにより FP32 SIMT Core で行列積を計算した場合と同程度まで計算精度が向上することが実験的に確認された (図 1)．一方で，このように計算することで，TensorCore

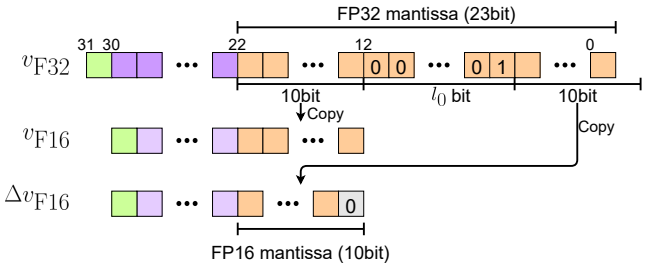

図 5: 式 7-8 での仮数部の移動（式 7 では RZ と丸めが行われると仮定）.

へ 0 を入力するためのレジスタやその初期化コスト，FP32 SIMT Core による加算コストの増加が発生する.

3.3 Δv_{F16} 計算におけるアンダーフローの抑制

次に式 8 におけるアンダーフローに注目する．この計算は値の近い 2 つの数の引き算であるため，指数部が小さい場合はアンダーフローが起こる．そこで，この計算におけるアンダーフロー及び漸進アンダーフローの起こりうる確率について考える.

定数として FP16 の指数部バイアス $b_{\mathrm{F16}} = 15$，及び FP16，FP32 の仮数部長 $l_{\mathrm{F16}} = 10, l_{\mathrm{F32}} = 23$ を定義する．また，簡単のため式 7 で行われる丸めが RZ であると仮定する．この場合，v_{F16} の仮数部は v_{F32} の仮数部の MSB から 10bit，Δv_{F16} の仮数部は v_{F32} の仮数部の $(10 + l_0 + 1)$bit 目から 10bit が格納される（図 5）．すなわち，式 8 により Δv_{F16} の指数部は v_{F32} の指数部から $l_0 + l_{\mathrm{F16}} + 1$ を引いた値となり，この指数部の値を用いてアンダーフローが起こるかを判定することができる.

今 v_{F32} の指数部の値がバイアスなしで e_v であるとする．すなわち $v + \mathrm{F32}$ は $v_{\mathrm{F32}} = 1.m_{22}m_{21}\cdots m_0 \times 2^{e_v}$ と表される．はじめに式 8 において漸進アンダーフローを起こす確率とアンダーフローを起こす確率の和 $P_{u+gu}(e_v)$ を考える．FP16 の正規化数で表される最小の数は $2^{-b_{\mathrm{F16}}}$ である．したがって，式 8 で漸進アンダーフローまたはアンダーフローが起こる条件は式 9 で表される.

$$e_v - (l_0 + l_{\mathrm{F16}} + 1) \leq -b_{\mathrm{F16}} \tag{9}$$

ここで，$l_0 = n, (n \in Z)$ となる確率 $P(l_0 = n)$ は，仮定 1 の元で式 10 で表される.

$$P(l_0 = n) = \begin{cases} 0 & (n < 0) \\ \left(\frac{1}{2}\right)^{n+1} & (0 \leq n < l_{\mathrm{F32}} - l_{\mathrm{F16}}) \\ \left(\frac{1}{2}\right)^{l_{\mathrm{F32}} - l_{\mathrm{F16}}} & (l_{\mathrm{F32}} - l_{\mathrm{F16}} \leq n) \end{cases} \tag{10}$$

これを用い，式 9 を満たす確率，すなわち $P_{u+gu}(e_v)$ は式 11 で表される.

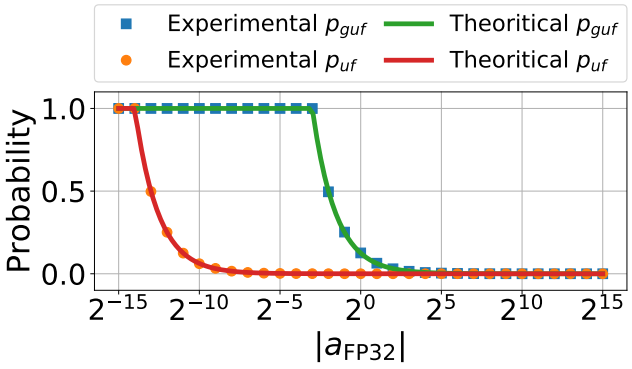

図 6: Δv_{F16} が漸進アンダーフロー及びアンダーフローを起こす確率．$P_{\mathrm{gu}}(v_e)$ は漸進アンダーフローを起こす確率，$P_{\mathrm{u+gu}}(v_e)$ は漸進アンダーフローもしくはアンダーフローを起こす確率である.

$$P_{u+gu}(e_v) = \sum_{i=0}^{l_{\mathrm{F32}} - l_{\mathrm{F16}} - 1} P(l_0 = e_v + b_{\mathrm{F16}} - l_{\mathrm{F16}} - 1 + i) \tag{11}$$

次にアンダーフローが起こる確率のみを考える．非正規化数で表される FP16 の最小値は $2^{-(b_{\mathrm{F16}} + l_{\mathrm{F16}})}$ であるため，アンダーフローが起こる条件は式 12 で表される.

$$\begin{aligned} & e_v + l_{\mathrm{F16}} - (l_0 + l_{\mathrm{F16}} + 1) \leq -b_{\mathrm{F16}} \\ \Rightarrow\ & e_v - (l_0 + 1) \leq -b_{\mathrm{F16}} \end{aligned} \tag{12}$$

これを満たす確率 P_{u} は，同様の議論により式 13 で表される.

$$P_u(e_v) = \sum_{i=0}^{l_{\mathrm{F32}} - l_{\mathrm{F16}} - 1} P(l_0 = e_v + b_{\mathrm{F16}} - 1 + i) \tag{13}$$

$P_{\mathrm{u}}(v_e)$, $P_{\mathrm{u+gu}}(v_e)$ および実験的にそれらを求めた値を表したものが図 6 である．これより，例えば $|v_{\mathrm{F32}}|$ が 2^0 程度の大きさでも式 8 で漸進アンダーフローが起こることが分かる.

そこで，漸進アンダーフロー及びアンダーフローが起こる確率を減らすため，式 14 に示すように式 8 において toF16 を行う前に指数部に $l_{\mathrm{F16}} + 1 = 11$ 加算する.

$$\Delta v_{\mathrm{F16}} \leftarrow \mathrm{toFP16}\left(\left(v_{\mathrm{F32}} - \mathrm{toFP32}\left(v_{\mathrm{F16}}\right) \times 2^{11}\right)\right) \tag{14}$$

本稿では 7 及び式 14 により v_{F32} を保持する手法を単に halfhalf，式 7 及び式 8 により保持する手法を Markidis らの halfhalf と呼ぶ.

これにより，式 1-5 の行列積は式 15-19 と置き換えられる.

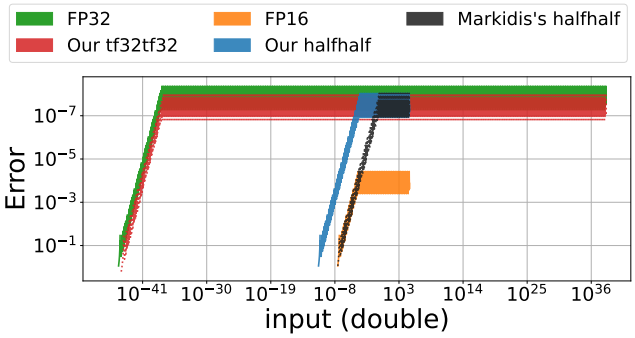

図 7: halfhalf，tf32tf32，FP32，FP16，Markidis らの halfhalf の表現精度と表現範囲．横幅が広いほどより大きな範囲を表現可能であることを表し，Error が小さいほど表現精度が良いことを表す．

$$\mathbf{A}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\mathbf{A}_{\mathrm{F32}}\right) \tag{15}$$
$$\Delta\mathbf{A}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\left(\mathbf{A}_{\mathrm{F32}} - \mathrm{toF32}\left(\mathbf{A}_{\mathrm{F16}}\right)\right) \times 2^{11}\right) \tag{16}$$
$$\mathbf{B}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\mathbf{B}_{\mathrm{F32}}\right) \tag{17}$$
$$\Delta\mathbf{B}_{\mathrm{F16}} \leftarrow \mathrm{toF16}\left(\left(\mathbf{B}_{\mathrm{F32}} - \mathrm{toF32}\left(\mathbf{B}_{\mathrm{F16}}\right)\right) \times 2^{11}\right) \tag{18}$$
$$\begin{aligned}\hat{\mathbf{C}}_{\mathrm{F32}} \leftarrow\ & \mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} \\ & + \left(\Delta\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} + \mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}\right)/2^{11} \\ & + \left(\Delta\mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}\right)/2^{22}\end{aligned} \tag{19}$$

ところで，NVIDIA Ampere アーキテクチャより TF32 (Tensor Float) と呼ばれるデータ型を TensorCore への入力として用いることが可能である．TF32 は指数部 8bit，仮数部 10bit からなり，指数部が FP32 と同じ bit 長である．このため，式 7 及び式 14 において FP16 の代わりに TF32 を用いることで式 14 でのアンダーフロー確率をさらに下げることが可能である．本稿ではこれを tf32tf32 と呼ぶ．現在 FP32 から TF32 への変換には `cvt.rna.tf32.fp32` 命令による RNA を用い，その仮数部長期待値は 22.75bit と計算される（3.1）．

この様に FP32 変数を halfhalf，Markidis らの halfhalf，tf32tf32 で保持した場合の表現可能範囲と表現精度を実験的に調査したものが図 7 である．halfhalf は Markidis らの halfhalf と比較して表現可能範囲が広く，tf32tf32 では FP32 と同程度の表現範囲を持つことが分かる．

3.4 $\Delta\mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}$ 項の無視

式 19 の $\Delta\mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}$ 項は $\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}}$ に対して 2^{-22} 程度の値となる．これは FP32 の仮数部 23bit の LSB のみに影響を与える値であり，$\Delta\mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}$ による精度補正能力は無視できると考えられる．そこで，この項を無視し式 19 を式 20 に置き換えて計算する．

$$\begin{aligned}\hat{\mathbf{C}}_{\mathrm{F32}} \leftarrow\ & \mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} \\ & + \left(\Delta\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} + \mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}\right)/2^{11}\end{aligned} \tag{20}$$

これにより必要な行列積の回数は Markidis らの手法と比較して 75%に抑えられる．

Implementation	TensorCore	Error Correction
cutlass_tf32tf32	TF32-TC	YES
cutlass_fp16fp16	FP16-TC	YES
cublas_tf32tc	TF32-TC	NO
cublas_fp16tc	FP16-TC	NO
cublas_simt(FP32)	Not used	NO

表 2: 評価に用いた単精度行列積実装．cutlass が本研究の精度補正計算を実装したものであり，cublas が reference のためのものである．cublas_simt では TensorCore は用いず，CUDA SIMT Core の FP32 演算器ので計算が行われる．

3.5 CUTLASS への実装

NVIDIA CUTLASS[*2]は NVIDIA が開発してオープンソースの行列演算ライブラリである．CUTLASS は C++ で書かれており，template 引数によりコンパイル時に行列積計算のカーネルを切り替えることが可能である．上記の TensorCore の accumulation の丸め改善，指数部シフトによるアンダーフロー抑制，微小項の無視を行い，CUTLASS への実装を行った．この実装では丸め改善は $\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}}$ についてのみ行っており，計算量削減のため $\Delta\mathbf{A}_{\mathrm{F16}}\mathbf{B}_{\mathrm{F16}} + \mathbf{A}_{\mathrm{F16}}\Delta\mathbf{B}_{\mathrm{F16}}$ 計算時には行っていない．

4. 実験

実験では表 2 に示す 5 種類の実装の比較を行った．計算には NVIDIA A100 SXM4 40GB[5]，CUDA 11.3 を用いた．

4.1 計算精度の評価

精度の評価は乱数行列を用いて行った．評価は入力行列を単精度で生成し，行列積計算 $\mathbf{C} = \mathbf{A}_{\mathrm{F32}} \times \mathbf{B}_{\mathrm{F32}}$ を行う．8 通りのシードにより生成した行列に対し式 6 による評価を行い，その平均を評価値として用いた．halfhalf は保持できる指数部の範囲が小さく，図 7 に示すとおり表現範囲が FP32 と比較して狭い．この影響も評価するため，入力行列の指数部の範囲ごとに評価を行う．ここで入力行列のクラス $\mathrm{exp\_rand}(a,b) \in \mathrm{FP32}^{M\times N}\ \ (a,b \in Z)$ を，その全ての要素の指数部が (a,b) から一様ランダムに選択され、仮数部の 23bit が $[0,\ 1]$ から一様ランダムに選択される行列と定義する．

この評価では次の 3 通りの $\mathrm{exp\_rand}(m,n)$ を入力行列として評価を行う．図 7 より次の性質が分かる．

- exp_rand(-15, 15)：
  全要素がおおよそ $(10^{-5}, 10^{5})$ の範囲にある．この場

[*2] https://github.com/NVIDIA/cutlass．バージョン 2.5 をもとに移植を行った．

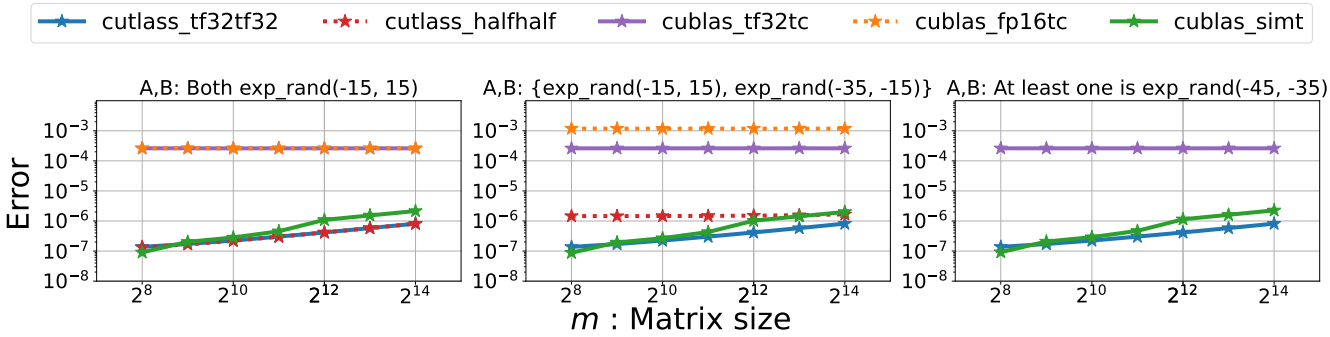

図 8: 入力行列の指数部の範囲による行列積の精度の評価．exp_rand(m,n) は各要素の指数部を (m,n) の一様乱数で選んだ行列を表す．仮数部 23bit も一様乱数から選ぶ．このとき，exp_rand$(-15,15)$ は全ての様をについて halfhalf で表した際にどちらの FP16 変数も正規化数となる行列，exp_rand$(-35,-15)$ は少なくとも一方の halfhalf は非正規化数となる行列，exp_rand$(-45,-35)$ は指数部を保持できず 0 となる行列である．**左**：$\mathbf{A},\mathbf{B}$ のどちらも exp_rand$(-15,15)$ とした場合の行列積の精度．**中央**：$\mathbf{A},\mathbf{B}$ のどちらか一方を exp_rand$(-15,15)$，もう一方を exp_rand$(-35,-15)$ とした場合の行列積の精度．**右**：$\mathbf{A},\mathbf{B}$ の少なくともどちらか一方を exp_rand$(-45,-35)$ とした場合の行列積の精度．

合 halfhalf の v も Δv も正規化数となる．

- exp_rand(-35, -15)：
  全要素がおおよそ $(10^{-12}, 10^{-5})$ の範囲にある．この場合 halfhalf の v と Δv の少なくとも一方は非正規化数となる．

- exp_rand(-45, -35)：
  全要素がおおよそ $(10^{-15}, 10^{12})$ の範囲にある．この場合 halfhalf では指数部を保持できず 0 となる．

これらの行列の積 $\mathbf{A}\times\mathbf{B}$ を計算した際の計算精度が図 8 である．$\mathbf{A}$ と $\mathbf{B}$ の両方が exp_rand(-15, 0) の場合は cutlass_halfhalf は cublas_simt とほぼ同じ精度で計算が行われていることが分かる．一方で，$\mathbf{A}$ と $\mathbf{B}$ の少なくとも一方が exp_rand(-35, -15) である場合は cublas_simt と比較して計算精度が劣化した．また，少なくとも一方が exp_rand(-45, -35) の場合は入力行列の値を保持できないため，正しく計算が行われなかった．これらの結果から，halfhalf を用いて行列積計算を行う場合は指数部に対するスケールシフトを行い，アンダーフローを抑制する必要があることが分かる．一方で，tf32tf32 ではどの指数部範囲でも FP32 SIMT Core を計算した場合と同程度の精度で計算が行われていることが確認された．

4.2 計算性能の評価

計算性能の評価では，いくつかの大きさの入力行列に対する行列積を計算し，その実行時間から計算性能（Flop/s）を算出した（図 9）．多くの行列サイズに対して cutlass_halfhalf と cutlass_tf32tf32 が cublas_simt と比較して高速に計算しており，cutlass_halfhalf では最大 45TFlop/s、cutlass_tf32tf32 では 29TFlop/s の性能が確認された．NVIDIA A100 の FP32 の理論ピーク性能は 19.5TFlop/s[5] であるため，この性能は FP32 SIMT Core のみを用いた場合では到達し得ない性能である．

5. 結論

本研究では TensorCore を用いた精度補正単精度行列積演算の計算精度の向上を行った．いくつかの仮定のもとで，入力の FP32 を 2 つの FP16 で分割して保持した場合の保存される仮数部の期待値を理論的に算出し，この分割が精度劣化の原因ではないことを示した．精度の劣化は TensorCore 内部での丸め処理等の省略であると考え，これを避けることで計算精度の向上を達成した．また，既存手法と比較してより小さい指数部を扱える指数部のリスケールや TF32 の利用による計算可能領域の拡大を達成した．この上で精度へ与える影響の小さい補正計算を削除することで計算量を落とし，NVIDIA CUTLASS へ移植し，その計算精度と計算性能の評価を行った．結果，TF32 TensorCore を用いた場合は FP32 と同程度の範囲の指数部を扱え，FP32 SIMT Core と同程度の精度でより高速に計算できることが確認された．入力行列の指数部が特定の範囲に入っている場合には FP16 TensorCore を用いた精度補正行列積により，FP32 SIMT Core と同程度の精度で計算を行え、FP32 SIMT Core や TF32 TensorCore を用いた場合より高速に計算できることが確認された．CUTLASS は深層学習用ライブラリ cuDNN の内部実装としても用いられており，このような単純な GEMM 以外への適用も可能であると考えられる．今後はこのような様々な計算に本精度補正手法を適用し，精度の劣化の伴わない高速なソフトウェア実装を目指す．

謝辞

本研究は JSPS 科研費 JP18H03248,JP21H03447 の助成を受けたものである．本研究は，JST，CREST，JP-

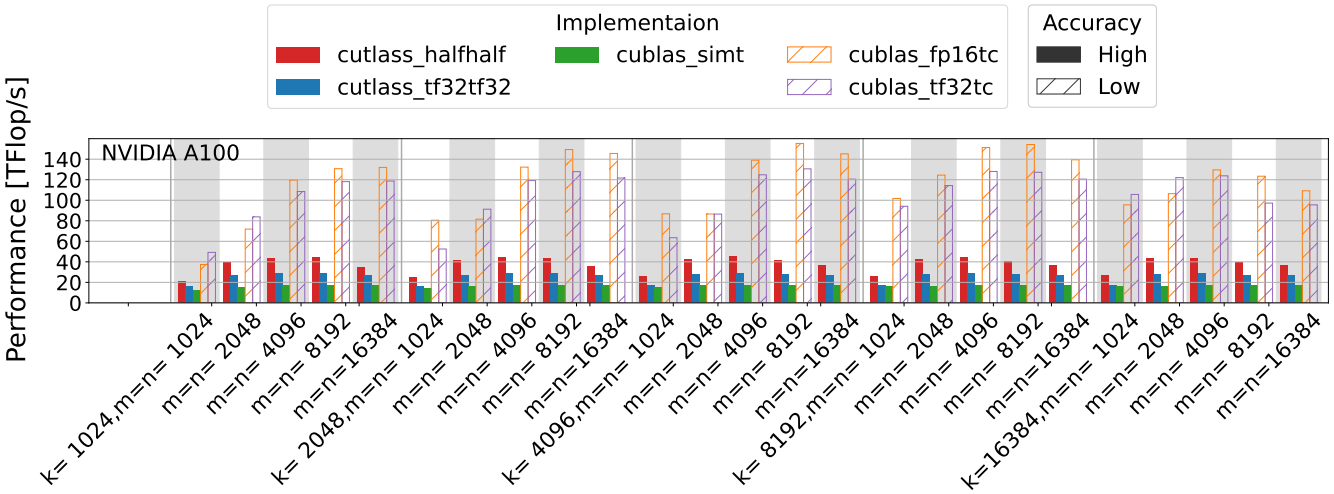

図 9: NVIDIA A100 での matmul-(m, n, k) の計算性能．塗りつぶしされた棒は FP32 SIMT Core を用いた場合と同程度の誤差で計算可能な実装である．斜線の棒は TensorCore を用いるが精度補正を行わない実装で，計算精度が FP16 と同程度の実装である．

MJCR19F5 の支援を受けたものである．本研究は，学際大規模情報基盤共同利用・共同研究拠点の支援による (課題番号: jh210024-NAHI)．

参考文献

[1] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. *SIAM Journal on Scientific Computing*, 42(3):C124–C141, January 2020. Publisher: Society for Industrial and Applied Mathematics.

[2] R. Carrasco, R. Vega, and C. A. Navarro. Analyzing GPU Tensor Core Potential for Fast Reductions. In *2018 37th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, November 2018. ISSN: 1522-4902.

[3] IBM Corporation. IBM Power Systems Announces POWER10 Processor, August 2020.

[4] NVIDIA Corporation. NVIDIA A100 TENSOR CORE GPU.

[5] NVIDIA Corporation. NVIDIA AMPERE GA102 GPU Architecture V1.

[6] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 46–57, New York, NY, USA, June 2019. Association for Computing Machinery.

[7] Massimiliano Fasi, Nicholas J. Higham, Mantas Mikaitis, and Srikara Pranesh. Numerical Behavior of the NVIDIA Tensor Cores, April 2020. Issue: 2020.10 Number: 2020.10.

[8] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 278–291, New York, NY, USA, February 2021. Association for Computing Machinery.

[9] J. S. Firoz, A. Li, J. Li, and K. Barker. On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, September 2020. ISSN: 2643-1971.

[10] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613, November 2018.

[11] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv:1804.06826 [cs]*, April 2018. arXiv: 1804.06826.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, and et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, June 2017. Association for Computing Machinery.

[13] Arm Ltd. Developments in the Arm A-Profile Architecture: Armv8.6-A.

[14] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. March 2018.

[15] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Toshiyuki Imamura. DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, Lecture Notes in Computer Science, pages 230–248, Cham, 2020. Springer International Publishing.

[16] Katsuhisa Ozaki, Takeshi Ogita, Shin' ichi Oishi, and Siegfried M. Rump. Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numerical Algorithms*, 59(1):95–118, January 2012.

[17] Md Aamir Raihan, Negar Goli, and Tor Aamodt. Modeling Deep Learning Accelerator Enabled GPUs. *arXiv:1811.08309 [cs]*, February 2019. arXiv: 1811.08309.

[18] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering*, 88:106848, December 2020.