

Tensor コアを用いた TSQR の GPU 実装

大友 広幸^{1,a)} 横田 理央^{1,b)}

概要: $m \gg n$ な $m \times n$ の行列に対する QR 分解アルゴリズムとして Tall-Skinny QR (TSQR) がある. TSQR では分割した入力行列に対して QR 分解を行うことで, 高速かつ数値的安定性を保ったままに QR 分解を行うことができる. 本研究では NVIDIA Volta アーキテクチャ世代より搭載された混合精度行列積演算回路である Tensor コアを用いて TSQR を実装し, 計算精度, 計算速度およびメモリ使用量の評価を行った. その結果, NVIDIA が開発を行っている cuSOLVER による QR 分解に対して, 本実装では高速かつ少ないメモリ使用量での QR 分解を行うことに成功した.

TSQR using TensorCore

1. はじめに

行列の低ランク近似は次元削減, データ圧縮, 計算量削減, ノイズ除去の要素技術として広く用いられている. 先行研究では, 特異値分解を用いた行列の低ランク近似アルゴリズムとして, あらかじめ定められたランク k での低ランク近似を乱数行列を用いることを行う Randomized SVD が考案されている [1]. Randomized SVD では $m \gg n$ である Tall Skinny な行列 $A \in \mathbb{R}^{m \times n}$ に対する QR 分解を必要とする. このような Tall Skinny な行列に対する QR 分解アルゴリズムとして TSQR (Tall-Skinny QR) [2] がある. TSQR では高い数値的安定性を保ったまま Tall Skinny な行列に対する QR 分解を少ないメモリで高速に行うことができる. さらに, 近年では大規模計算環境における TSQR の研究も行われている [3].

実際の計算においては半精度 (FP16) などの低精度変数を用いることで計算速度の向上やメモリ消費量の削減が期待される. 近年では NVIDIA Volta アーキテクチャ世代より混合精度行列演算回路である Tensor コアが導入された. Tensor コアは 2 つの 4×4 半精度行列 A, B と半精度または単精度の 4×4 行列 C の積和 $A \times B + C$ を計算する回路である. Tensor コアでは内部の足しこみ処理を高精度で行うことで, 数値計算による誤差の蓄積を抑えた上で高速

に行列積和を計算することができる. 行列の低ランク近似は高い計算精度が必要とされないことから, 低精度変数と Tensor コアを用いることで計算の高速化と省メモリ化の恩恵を十分に享受できると考えられる.

本研究では Tensor コアを用いた Householder 変換 [4] による QR 分解及び TSQR を実装し, その計算精度, 計算速度, メモリ使用量の評価を行った.

2. 背景

2.1 QR 分解

QR 分解では $m \leq n$ を満たす行列 $A \in \mathbb{R}^{m \times n}$ を $A = Q'R'$ を満たすような直交行列 $Q' \in \mathbb{R}^{m \times m}$ と上三角行列 $R' \in \mathbb{R}^{m \times n}$ に分解する. 行列 R' は上三角行列であるため, $\exists Q \in \mathbb{R}^{m \times n}, Q_0 \in \mathbb{R}^{m \times m-n}, R \in \mathbb{R}^{n \times n}$ に対し

$$\begin{aligned} A &= Q'R' \\ &= [Q|Q_0] \begin{bmatrix} R \\ 0 \end{bmatrix} \\ &= QR \end{aligned}$$

が成り立つ. Q, R を thin Q , thin R と呼び, この分解を Thin QR 分解と呼ぶ. 本論文では Thin QR 分解を単に QR 分解と呼ぶ.

2.1.1 Householder 変換を用いた QR 分解

Householder 変換では単位行列 $I \in \mathbb{R}^{m \times m}$, ベクトル $u \in \mathbb{R}^m$ を用いて式 (1)

$$H = I - 2 \frac{uu^T}{|u|^2} \quad (1)$$

¹ 東京工業大学
Tokyo Institute of Technology
a) ootomo.h@rio.gsic.titech.ac.jp
b) riyoikota@gsic.titech.ac.jp

で得られる Householder 行列 H により $x \in \mathbb{R}^n$ に対する原点を通る u に直交する直線での鏡映 Hx を得る. x を $y = [\pm|x| \ 0 \ 0 \cdots 0]^T \in \mathbb{R}^n$ に映す Householder 行列 H は

$$u = x \mp y \quad (2)$$

とすることで得る (複号同順). この Householder 変換を入力行列 A が上三角行列となるように各列に対し行うことで数値的に安定な QR 分解を行う [2].

2.1.2 Householder 変換を用いた QR 分解アルゴリズム

Householder 変換を用いた QR 分解は, アルゴリズム 1 に示す Householder 行列 H の生成と行列積で構成される.

アルゴリズム 1 Householder 変換を用いた QR 分解

Require: $m, n \in \mathbb{N}, A \in \mathbb{R}^{m \times n}$

Ensure: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$

```

1:  $Q' \leftarrow I_m$ 
2:  $R' \leftarrow A$ 
3: for  $i \leftarrow 0$  to  $n - 1$  do
4:    $u \leftarrow [0 \ \cdots \ 0 \ R'_{i,i} \ \cdots \ R'_{m-1,i}]^T$ 
5:    $u_i \leftarrow u_i + \text{sign}(u_i)|u|$ 
6:    $H \leftarrow I - 2 \frac{uu^T}{|u|^2}$ 
7:    $R' \leftarrow HR'$ 
8:    $Q' \leftarrow HQ'$ 
9: end for
10:  $Q \leftarrow \text{Reshape}(Q'^T, m, n)$ 
11:  $R \leftarrow \text{Reshape}(R', n, n)$ 

```

表記

$\text{sign}(v)$
 $v \in \mathbb{R}$ の符号
 $\text{Reshape}(M, N_m, N_n)$
行列 M の $(1, 1)$ 要素から大きさ (N_m, N_n) の行列を切り出す

2.2 TSQR

$m \gg n$ な Tall Skinny な行列 $A \in \mathbb{R}^{m \times n}$ に対する効率的な QR 分解アルゴリズムとして TSQR が考案された. TSQR では A を行方向に分割し, 分割されたそれぞれの行列に対し QR 分解を繰り返し行うことで少ないメモリ使用量で QR 分解を行うことができる. このアルゴリズムは次式

$$A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \xrightarrow{\text{A}_0, \text{A}_1 \text{ に対して QR 分解}} \begin{bmatrix} Q_0 R_0 \\ Q_1 R_1 \end{bmatrix} = \begin{bmatrix} Q_0 & 0 \\ 0 & Q_1 \end{bmatrix} \begin{bmatrix} R_0 \\ R_1 \end{bmatrix}$$

が成り立つことを再帰的に用いることで示される.

Householder 分解を用いた TSQR では Householder 分解を用いた QR 分解と同様に無条件での数値的安定性がある.

2.3 NVIDIA GPU による半精度演算

NVIDIA GPU で採用されている半精度浮動少数 (FP16) は, IEEE 754 Binary16 で, 仮数部 5bit, 指数部 10bit, 符号部 1bit からなる. NVIDIA GPU では半精度を使用することでメモリ消費量を半分にするだけでなく, 2つの FP16 変数を同時に処理する SIMD 命令が備わっており, これにより計算性能が向上することが報告されている [5]. SIMD 命令は ADD, SUB, MUL の他, FMA (Fused Multiply-Add) 命令などが備わっている.

2.4 Tensor コア

NVIDIA Volta アーキテクチャより混合精度行列積演算回路である Tensor コアが搭載された. Tensor コアでは行列 $A, B \in \text{FP16}^{4 \times 4}, C, D \in \text{FP16}/\text{FP32}^{4 \times 4}$ に対し行列積和

$$D \leftarrow A \times B + C$$

を 1 Warp (32 Threads) が協調して計算する. Tensor コアではこの足しこみ処理を精度を損なうことなく行うことができ, これにより単純に FP16 で計算した場合に比べ計算精度の劣化が少ないという特長がある.

3. TSQR の実装

表 1 に示す 4 通りの実装を行った. それぞれの実装は入力行列 $A \in \mathbb{R}^{m \times n}$ for $1 \leq m, 1 \leq n \leq 32$ に対する QR 分解を行う.

名称	入出力型	Tensor コア
TSQR-FP32-TC	float	使用
TSQR-FP16-TC	half	使用
TSQR-FP32	float	非使用
TSQR-FP16	half	非使用

表 1 実装一覧

3.1 TSQR の計算手順

本実装では次の手順で計算を行う.

(1) 入力行列の分割

入力行列 $A \in \mathbb{R}^{m \times n}$ を

$$A = [A^{(0)}] = \begin{bmatrix} A_0^{(0)} \\ A_1^{(0)} \\ \vdots \\ A_{2^b-1}^{(0)} \end{bmatrix}, \text{ for } A_i^{(0)} \in \mathbb{R}^{m_i \times n} \quad (3)$$

のように 2^b 個の行列に分割する. この際 $\forall i, 16 \leq m_i < 32$ が成り立つように分割数 2^b を選択する. これを満たす b は

$$b = \max(0, \lceil \log_2 m \rceil - 5) \quad (4)$$

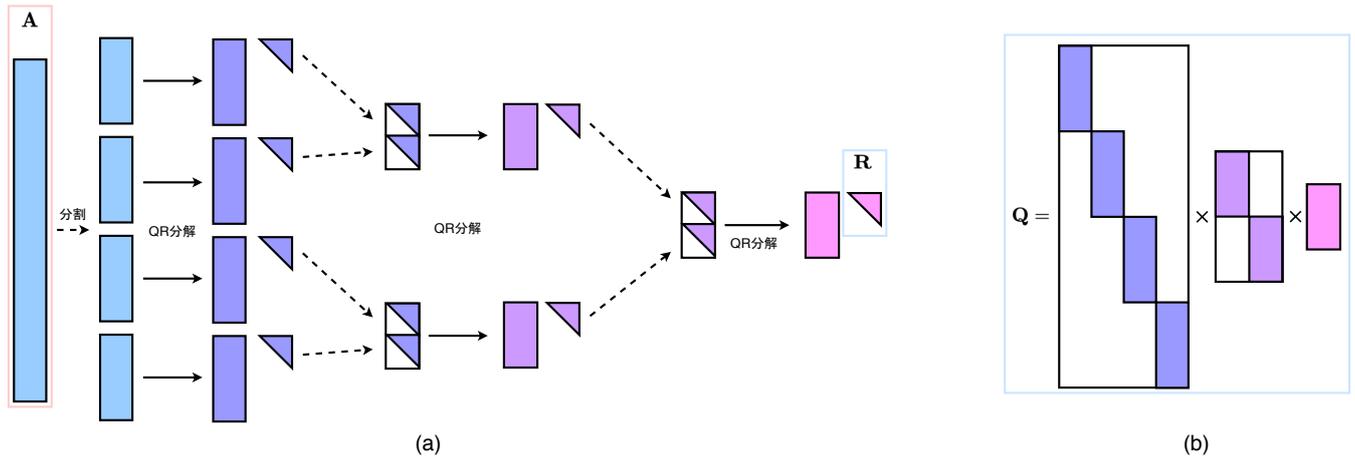


図 1 TSQR; 入力行列を 4 分割する TSQR の概要図 . (a) 分割された行列に対して QR 分解を行い, それを結合することを繰り返し目的の \mathbf{R} を計算する . (b)(a) で繰り返し行われる QR 分解の結果を用いて目的の \mathbf{Q} を計算する .

と計算され, m_i は

$$m_i = \left\lceil \frac{m \times (i+1)}{2^b} \right\rceil - \left\lceil \frac{m \times i}{2^b} \right\rceil \quad (5)$$

とすれば良い .

(2) \mathbf{R} の計算

$\mathbf{A}_i^{(k)}$ に対し Householder 変換を用いた QR 分解を行い

$$\mathbf{Q}_i^{(k)}, \mathbf{R}_i^{(k)} \xleftarrow{\text{QR 分解}} \mathbf{A}_i^{(k)} \quad (6)$$

を $\leq i \leq 2^{b-1-k}-1$ に対して行う . $\mathbf{A}_j^{(k+1)} = \begin{bmatrix} \mathbf{R}_{2j}^{(k)} \\ \mathbf{R}_{2j+1}^{(k)} \end{bmatrix}$ とし, $k = 0, \dots, b-2$ と行うことで $\mathbf{R} = \mathbf{R}_0^{(b-1)}$ を得る .

(3) \mathbf{Q} の計算

$\mathbf{Q}_0^{(b-1)}$ を

$$\mathbf{Q}_0^{(b-1)} = \begin{bmatrix} \hat{\mathbf{Q}}_0^{(b-1)} \\ \hat{\mathbf{Q}}_1^{(b-1)} \end{bmatrix}, (\hat{\mathbf{Q}}_0^{(b-1)}, \hat{\mathbf{Q}}_1^{(b-1)} \in \mathbb{R}^{n \times n})$$

と分割する . $0 \leq i \leq 2^{b-k}-1$ を満たすすべての i に対し

$$\hat{\mathbf{Q}}_i^{(k-1)} = \mathbf{Q}_i^{(k-1)} \hat{\mathbf{Q}}_i^{(k)} \quad (7)$$

を $k = b-1, \dots, 1$ と繰り返し計算することで

$$\mathbf{Q} = \begin{bmatrix} \hat{\mathbf{Q}}_0^{(0)} \\ \hat{\mathbf{Q}}_1^{(0)} \\ \vdots \\ \hat{\mathbf{Q}}_{2^{b-1}-1}^{(0)} \end{bmatrix}$$

を得る .

式 (6) における QR 分解は, すべての i において独立に計算することができる . ここでの QR 分解では複数の行列

に対して同時に QR 分解を行う Batched QR 分解を実装しこれを用いた . 同様に, 式 (7) における行列積もすべての i に対して独立に計算できるため, これを並列して計算する Batched Matmul を実装し, これを用いた . 実装した TSQR を擬似コードを用いて記述するとアルゴリズム 2 となる .

アルゴリズム 2 TSQR

Require: $m, n \in \mathbb{N}, \mathbf{A} \in \mathbb{R}^{m \times n}$

Ensure: $\mathbf{Q} \in \mathbb{R}^{m \times n}, \mathbf{R} \in \mathbb{R}^{n \times n}$

- 1: $b = \max(0, \lceil \log_2 m \rceil - 5)$
- 2: $[\mathbf{A}^{(0)}] = \text{Split}(\mathbf{A}) = \{\mathbf{A}_0^{(0)}, \mathbf{A}_1^{(0)}, \dots, \mathbf{A}_{2^b-1}^{(0)}\}$
- 3: for $i \leftarrow 0$ to $b-1$ do
- 4: $[\mathbf{Q}^{(i)}], [\mathbf{R}^{(i)}] = \text{BatchedQR}([\mathbf{A}^{(i)}])$
- 5: if $i \neq b-1$ then
- 6: $[\mathbf{A}^{(i+1)}] = \left\{ \begin{bmatrix} [\mathbf{R}^{(i)}]_{2k} \\ [\mathbf{R}^{(i)}]_{2k+1} \end{bmatrix} \text{ for } k = 0.. \lfloor 2^{b-i-2} \rfloor \right\}$
- 7: else
- 8: $\mathbf{R} = [\mathbf{R}^{(i)}]_0$
- 9: end if
- 10: end for
- 11: $[\hat{\mathbf{Q}}^{(b-1)}] = \text{Split}([\mathbf{Q}^{(b-1)}]_0) = \{\hat{\mathbf{Q}}_0^{(b-1)}, \hat{\mathbf{Q}}_1^{(b-1)}\}$
- 12: for $i \leftarrow b-1$ to 1 do
- 13: $[\hat{\mathbf{Q}}^{(i-1)}] = \text{BatchedMatmul}([\mathbf{Q}^{(i-1)}], [\hat{\mathbf{Q}}^{(i)}])$
- 14: end for
- 15: $\mathbf{Q} = \left[[\hat{\mathbf{Q}}^{(0)}]_0^T [\hat{\mathbf{Q}}^{(0)}]_1^T \dots [\hat{\mathbf{Q}}^{(0)}]_{2^b-1}^T \right]^T$

表記

$[\mathbf{X}]$	行列 \mathbf{X} の配列
$[\mathbf{X}]_i$	行列 \mathbf{X} の配列の第 i 要素
$\{\cdot\}$	配列の内包/外延表記
$\begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \end{bmatrix}$	行列 \mathbf{X}, \mathbf{Y} を縦に結合した行列

3.2 Tensor コアの使用について

3.2.1 Tensor コアの使用箇所

Tensor コアを使用する実装 (TSQR-FP32-TC, TSQR-FP16-TC) では次の 2 箇所 Tensor コアを使用した。

- (a) アルゴリズム 1 : 7, 8 行目の行列積。
- (b) アルゴリズム 2 : 15 行目の Batched 行列積。

Tensor コアを使用しない半精度での実装 TSQR-FP16 では, Tensor コアの代わりに SIMD FMA 関数 `_hfma2` を使用した行列積を用いた。

3.2.2 レジスタブッキング

Tensor コアを用いた行列積 $C \leftarrow A \times B$ は Tensor コアの API である WMMA API を用いて次の手順で行う (図 2)。

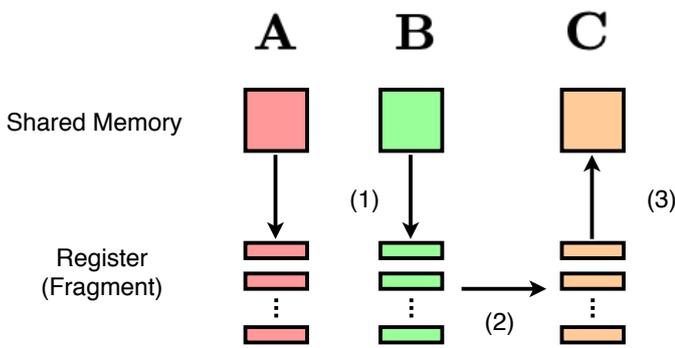


図 2 WMMA API を用いた行列積計算

- (1) Shared メモリにある A, B のデータをスレッドごとに Fragment と呼ばれる断片に分割しレジスタにコピー。
- (2) A, B それぞれの Fragment から Tensor コアを用いて C の Fragment を計算。
- (3) C の Fragment を Shared メモリにコピー。

このように WMMA API を用いる場合スレッドごとに Fragment という形でそれぞれの行列の一部をレジスタに保持する。この Fragment をアルゴリズム 1 の 7, 8 行目の行列 H の行列積で使い分回すことで Shared メモリへのアクセス回数を削減することができる。

4. 実験

次の構成で実験を行った。

- 入力行列 A は $[-1, 1]$ の一様乱数で初期化。
- cuSOLVER^{*1} の QR 分解関数群を用いた FP32 計算と比較。
- 各実験は 16 回行い, その平均をグラフに示した。

^{*1} <https://docs.nvidia.com/cuda/cusolver>

• 計算機:

- CPU: Intel Xeon CPU E5-2630 v3
- GPU: Tesla V100-PCIE-16GB
- RAM: 64GB

4.1 TSQR の計算精度の調査

Tensor コアを用いた QR 分解の計算精度を調査するため, 入力行列 $A \in \mathbb{R}^{m \times 16}$ に対し m を 2^{10} から 2^{25} と 2 倍ずつ変化させた場合の誤差率と直交性を求めた (図 3, 4)。ただし, 誤差率 E_{QR} はフロベニウスノルム $\|\cdot\|_F$ を用いて

$$E_{QR} = \frac{\|A - QR\|_F}{\|A\|_F} \quad (8)$$

と定義し, 直交性 I_{QR} は

$$I_{QR} = \frac{\|QQ^T - I\|_F}{\sqrt{n}} \quad (9)$$

と表される。FP16 実装における誤差率 E_{QR} 及び直交性 I_{QR} の評価は出力行列を FP32 にキャストし FP32 の行列積を用いて行った。

図 3 から TSQR-FP32-TC では TSQR-FP16, TSQR-FP16-TC に比べ誤差率が低く, 精度の劣化が抑えられていることがわかる。これにより Tensor コアを用いた FP32 での TSQR 実装は単純な FP16 実装や Tensor コアを用いた FP16 実装に比べ優位であることが確認された。

図 4 から FP32 で Tensor コアを用いた実装では FP16 で Tensor コアを用いた場合に比べ直交性の値が低く, より精度の高い Q が計算されていることが確認された。一方で TSQR-FP32 及び cuSOLVER での直交性は $m = 2^{21}$ を境に劣化しているのに対し, TSQR-FP32-TC/TSQR-FP16/TSQR-FP16-TC では直交性の劣化は確認されない。QR 分解で得られる行列 R は対角成分が大きくなる。一様乱数で初期化した行列に対する TSQR ではアルゴリズム 2 の 3 行目のループ 1 回につき $[R^{(i)}]$ の対角成分はおおよそ $\sqrt{2}$ 倍される (付録 A.1)。これによりループ回数が多いほど $[R^{(i)}]$ の対角成分と非対角成分の大きさの比が大きくなり, 足しこみの際などに計算誤差が蓄積されやすいと考えられる。TSQR-FP32-TC/TSQR-FP16/TSQR-FP16-TC では, 内部の計算に FP16 を用いることでそもそも精度が低く, 上記の計算誤差の影響は無視できるものであるため直交性の劣化が確認されないものと考えられる。

4.2 TSQR の計算速度の調査

Tensor コアを用いた QR 分解の計算精度を調査するため, 入力行列 $A \in \mathbb{R}^{m \times 16}$ に対し m を 2^{10} から 2^{25} と 2 倍ずつ変化させた場合の計算速度と計算性能の調査を行った (図 5, 6)。図 6 において cuSOLVER の計算性能の評価は cuSOLVER で用いられているアルゴリズムが不明なため演算量を計算できず行っていない。

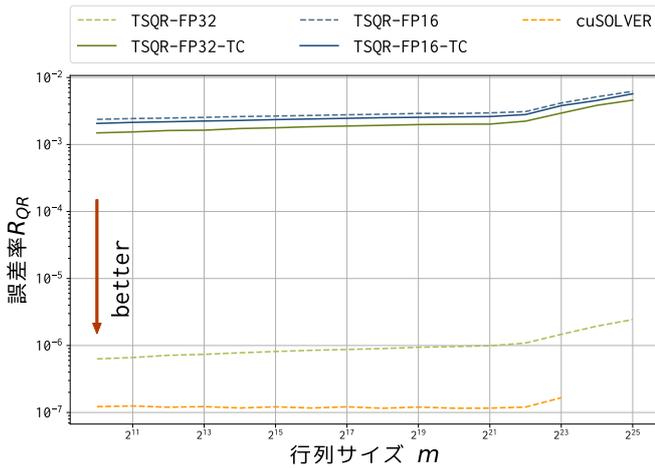


図 3 行列サイズと誤差率の関係

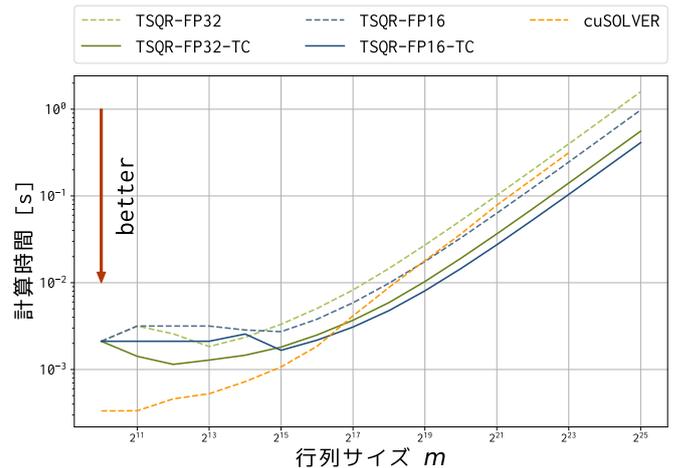


図 5 行列サイズと計算性能の関係

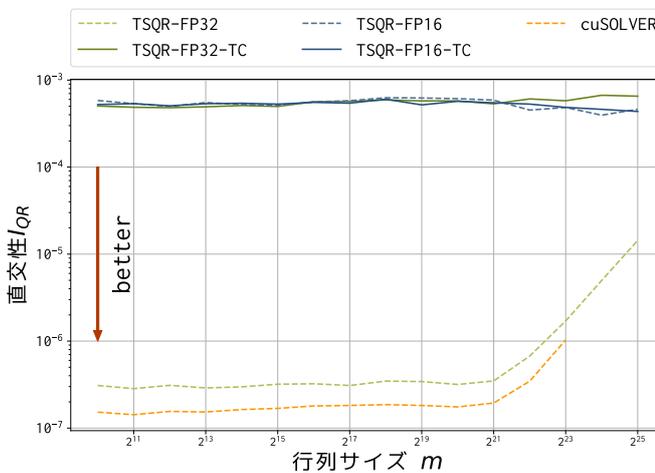


図 4 行列サイズと直交性 I_QR の関係

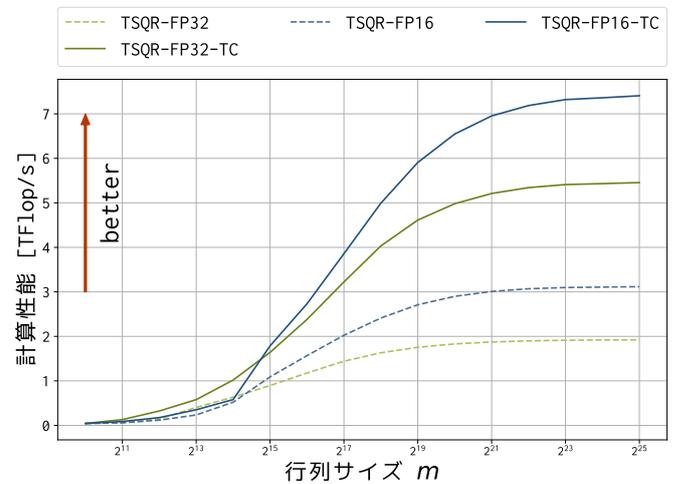


図 6 行列サイズと計算性能の関係

図 5 より $m \geq 2^{15}$ において Tensor コアを用いた実装 (TSQR-FP32-TC, TSQR-FP16-TC) が Tensor コアを用いた実装 (TSQR-FP32, TSQR-FP16) に比べ 2~3 倍高速であった。また, cuSOLVER との比較では $m \leq 2^{17}$ において TSQR-FP32-TC の方が高速であり, 最大 2.17 倍高速であることが確認された。

図 6 より TSQR-FP16-TC で最大 7.4TFlop/s, TSQR-FP32-TC で最大 5.4TFlop/s の計算効率を達成したことが確認された。

4.3 TSQR の作業用メモリ量

本実装および cuSOLVER で QR 分解を行う場合, Global メモリに作業用メモリを必要とする。入力行列 $A \in \mathbb{R}^{m \times 16}$ に対し m を 2^{10} から 2^{24} と 2 倍ずつ変化させた場合の作業用メモリの大きさを調査した (図 7)。Tensor コアを用いた FP32 実装である TSQR-FP32-TC では cuSOLVER と比較して 21.5%の作業用メモリで計算を行っていることが確認された。

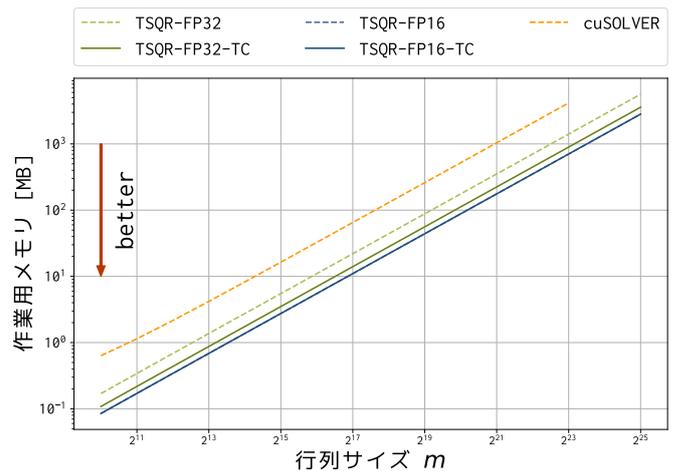


図 7 行列サイズと作業用メモリの関係

4.4 Q, R の計算時間の調査

入力行列 $A \in \mathbb{R}^{m \times 16}$ に対し m を 2^{10} から 2^{24} と 2 倍ずつ変化させた場合の Q, R の計算にかかる時間を計測した (図 8)。R の計算に必要な Tensor コアによる 16×16 の行列積の回数は $n = 16$ の場合 $96 \times (2^{b+1} - 1) \sim 96 \times 2^{b+1}$

回、 Q の計算に必要な回数は $2 \times (2^b - 1) \sim 1 \times 2^{b+1}$ 回となっており、その比率はおおよそ 96 : 1 であり、全体のおおよそ 1% が Q 計算での行列積である。図 8 より Q 計算の時間の割合はおおよそ 3% となっており、行列積の回数から導出した理論値より高い割合となっている。原因としては本実装の方式上 Q 計算で立ち上げるスレッド数が R 計算のスレッド数の半数となっており、 Q 計算におけるメモリアクセスの隠蔽率が低くなっていることが考えられる。

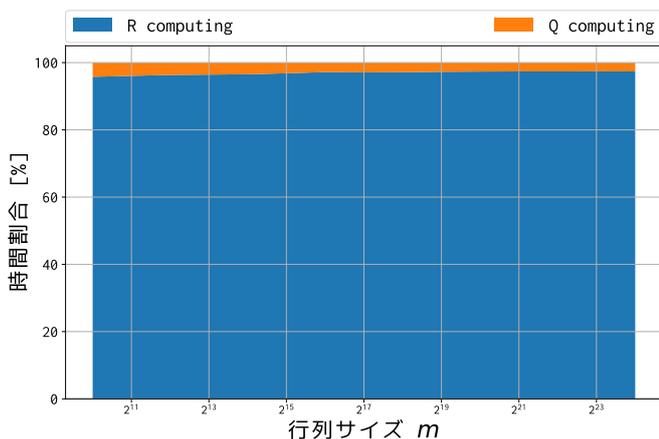


図 8 Q, R の計算時間の割合

4.5 Householder 行列を陽に作らなかった場合との比較

Householder 変換を用いた QR 分解ではアルゴリズム 1 の 6 行目の Householder 行列の作成を陽に行わず、7, 8 行目を

$$\mathbf{R}' \leftarrow \mathbf{R}' - 2 \frac{\mathbf{u}\mathbf{u}^T}{|\mathbf{u}|^2} \mathbf{R}' \quad (10)$$

$$\mathbf{Q}' \leftarrow \mathbf{Q}' - 2 \frac{\mathbf{u}\mathbf{u}^T}{|\mathbf{u}|^2} \mathbf{Q}' \quad (11)$$

と一般に計算する。本実装では Householder 行列 \mathbf{H} を陽に計算し \mathbf{R}', \mathbf{Q}' の更新を行っているが、このように陽に計算しなかった場合との比較を行った。入力行列 $\mathbf{A} \in \mathbb{R}^{m \times 16}$ に対し m を 2^{10} から 2^{25} と 2 倍ずつ変化させた場合の誤差率 E_{QR} , 直交性 I_{QR} , 計算時間を計測した (図 9, 10, 11)。

図 9 より、陽に Householder 行列を計算を行わない方が誤差率が低くなっていることが分かる。これは例えば式 (10) の場合、陽に Householder 行列 \mathbf{H} を計算する場合

$$\mathbf{R}'_{FP32} \leftarrow \mathbf{H}_{FP16} \mathbf{R}'_{FP16} \quad (12)$$

と右辺はすべて FP16 入力で計算するところを

$$\mathbf{R}'_{FP32} \leftarrow \mathbf{R}'_{FP32} + \left(-2 \frac{\mathbf{u}\mathbf{u}^T}{|\mathbf{u}|^2} \right)_{FP16} \mathbf{R}'_{FP16} \quad (13)$$

と右辺第 1 項を FP32 入力で計算することで誤差率を低く抑えられたと考えられる。

一方で図 10 より、Householder 行列 \mathbf{H} を陽に計算するが否かは直交性に影響をあまり与えないものと考えられる。

図 11 より陽に Householder 行列 \mathbf{H} を計算する場合のほうが高速に QR 分解を行っていることが分かる。式 (13) を WMMA API で計算する場合、 $\mathbf{H}_{FP16}, \mathbf{R}'_{FP16}$ の 2 つの行列を Fragment として Shared メモリから読み込むのに対し、式 (12) の場合、 $\mathbf{R}'_{FP32}, \left(-2 \frac{\mathbf{u}\mathbf{u}^T}{|\mathbf{u}|^2} \right)_{FP16}, \mathbf{R}'_{FP16}$ の 3 つの行列を読み込む必要がある。このメモリの読み込みの時間差に対して Householder 行列 \mathbf{H} を陽に作る時間の方が短かったため、陽に \mathbf{H} を計算する実装の方が高速であったと考えられる。

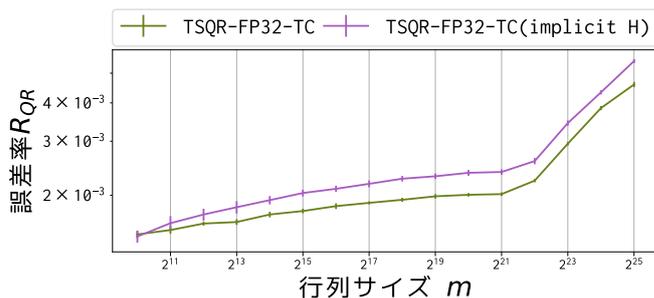


図 9 行列サイズと誤差率 E_{QR} の関係性

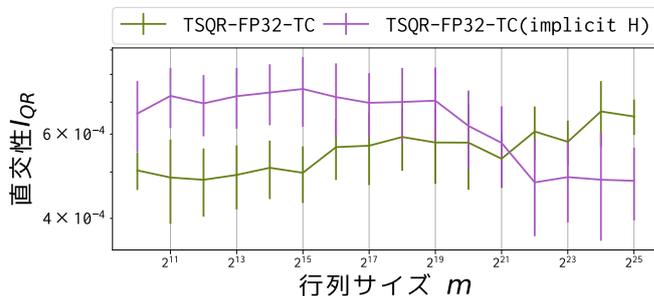


図 10 行列サイズと直交性 I_{QR} の関係性

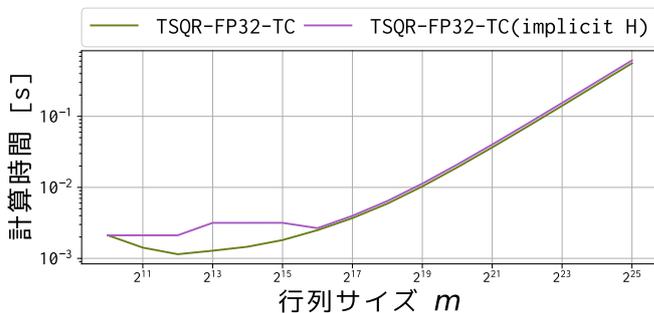


図 11 行列サイズと計算時間の関係性

5. おわりに

本研究では Tall Skinny な行列に対する QR 分解アルゴリズムである TSQR を Tensor コアを用いて実装し、その精度と計算精度の調査を行った。その結果次の 3 つの結論を得た。

- Tensor コアを用いた FP32 での TSQR 実装では FP16 での実装に比べ精度の劣化が抑えられた。
- Tensor コアを用いた FP32 での TSQR 実装では cu-SOLVER による QR 分解に比べ最大 2.17 倍高速に計算を行うことができた。
- Tensor コアを用いた FP32 での TSQR 実装では cu-SOLVER による QR 分解に比べ 21.5% の作業用メモリ使用量で計算を行うことができた。

今後の課題

Randomized SVD への応用

Randomized SVD で行列の低ランク近似を行う際に Tall Skinny な行列に対する QR 分解を行う必要がある。本研究で実装した Tensor コアを用いた TSQR を用いた Randomized SVD を実装し、低ランク近似の精度に与える影響の調査を行う。

汎用的な大きさの QR 分解への応用

パネルブロッキングを用いた QR 分解アルゴリズム [6] では入力行列を Tall Skinny な行列に分割し QR 分解を行うことで少ないメモリ消費量で QR 分解を行うことができる。本研究で実装した TSQR を用いることで汎用的な大きさの QR 分解においても高速に計算を行えることが期待される。

謝辞

本研究を行うにあたり北海道大学の深谷猛助教にご指導いただきましたことを深く感謝いたします。本研究は JST CREST JPMJCR19F5 の支援を受けたものである。本研究は JSPS 科研費 JP18H03248 の助成を受けたものである。

参考文献

- [1] N. Halko, P. G. Martinsson, J. A. Tropp *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Volume 53 Issue 2, May 2011 Pages 217-288
- [2] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou: *Communication-optimal parallel and sequential QR and LU factorizations: theory and practice*, arXiv:0806.2159v3 [cs.NA] 29 Aug 2008.
- [3] 深谷 猛: FX10 4800 ノードを用いた通信削減型 QR 分解アルゴリズムの性能評価, スーパーコンピューティング ニュース Vol.16, No.4, 2014
- [4] Lloyd N, Trefethen, David Bau Numerical linear algebra

SIAM 1997

- [5] 例えば <https://www.nvidia.com/content/apac/gtc/ja/pdf/2017/1040.pdf>
- [6] Alfredo Buttari, Julien Langou, Jakub Kurzak, Jack Don- garra *Parallel tiled QR factorization for multicore architectures*, PPAM'07 Proceedings of the 7th international conference on Parallel processing and applied mathematics P 639-648

付 録

A.1 R の対角成分

一様乱数行列に対する TSQR を考える。Householder 変換を用いた QR 分解では \mathbf{R} の上三角部分以外を 0 とするために対角成分の絶対値を増加させる。従って、アルゴリズム 2 の 6 行目で作られる行列 $[\mathbf{A}^{(i+1)}]_k = \begin{bmatrix} [\mathbf{R}^{(i)}]_{2k} \\ [\mathbf{R}^{(i)}]_{2k+1} \end{bmatrix}$ は $[\mathbf{R}^{(i)}]_{2k}, [\mathbf{R}^{(i)}]_{2k+1}$ の対角成分部分が他の要素に比べ大きい。 $[\mathbf{A}^{(i+1)}]_k$ の l 列目は \mathbf{a}_l は $[\mathbf{R}^{(i)}]_{2k}, [\mathbf{R}^{(i)}]_{2k+1}$ の対角成分を $d_{2ki,i}, d_{2k+1,i}$ とし、それらの非対角成分を無視すると

$$\mathbf{a}_l = \begin{bmatrix} * \\ \vdots \\ d_{2ki,i} \\ 0 \\ \vdots \\ 0 \\ * \\ \vdots \\ d_{2k+1,i} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (\text{A.1})$$

と書ける。入力行列が一様乱数行列なため $d_{2ki,i} = d_{2k+1,i} = d_i$ とすると、アルゴリズム 1 の 6 行目で作られる Householder 行列 \mathbf{H} は d_i に寄らず

$$\mathbf{H} \sim \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & & & \vdots \\ 0 & \cdots & \pm \frac{1}{\sqrt{2}} & \cdots & \pm \frac{1}{\sqrt{2}} & \cdots & * \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & & \pm \frac{1}{\sqrt{2}} & \cdots & \pm \frac{1}{\sqrt{2}} & \cdots & * \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & * & \cdots & * & \cdots & * \end{bmatrix} \quad (\text{A.2})$$

となる。これをアルゴリズム 1 の 7 行目のように \mathbf{R}' にかけると対角成分がおおよそ $\sqrt{2}d_i$ となるため、TSQR では QR 分解を繰り返すたびに対角成分がおおよそ $\sqrt{2}$ 倍となる。