TSQR on TensorCores

Hiroyuki Ootomo[1] , **Rio Yokota**[2]

[1]School of Computing, TokyoTech [2]Global Scientific Information and Computing Center, TokyoTech

https://github.com/enp1s0/tsqr-gpu

Tokyo Tech

Abstract

Tall-Skinny QR (TSQR) is an efficient algorithm for calculating the QR decomposition of $m \times n$ matrices where $m \gg n$, which is done by recursively performing QR decomposition on subdivided blocks of the tall and skinny matrix. Such operations are useful for low-rank approximation methods, which are replacing more and more dense linear algebra in both scientific computing and machine learning fields. The present work focuses on the implementation of this important algorithm on Tensor Cores, which are available on the latest NVIDIA GPUs. We evaluate the speed, accuracy, and stability of TSQR on TensorCores.

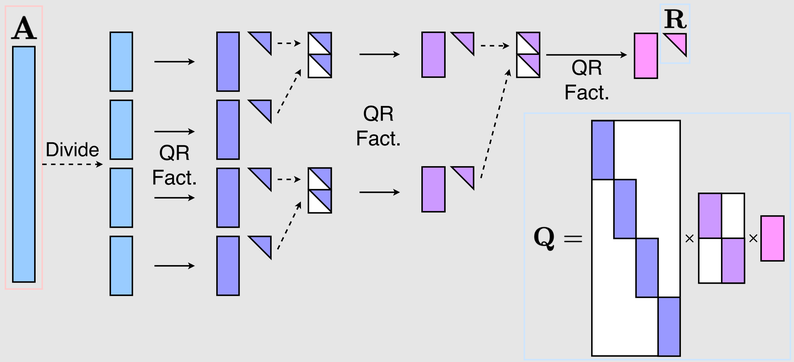
Test Environment

Machine
- Intel Xeon CPU E5-2630 v3 @ 2.40GHz x2
- NVIDIA Tesla V100-PCIE-16GB
- 64GB RAM
- Ubuntu 18.04
- CUDA 10.1

Input Matrix
- $m \times n$ matrix ($n = 16$ fixed)
- Randomized with $[-1, 1]$ uniform distribution

Comparison
- cuSOLVER (FP32, FP64)

TSQR on TensorCores



Algorithm 1. TSQR

1. Divide the input matrix **A**.
2. Calculate QR decomposition for each subdivided matrices to get **R**s and **Q**s.
3. Merge consecutive **R** blocks.
4. Repeat 2-3 until there is only one **R**.
5. Calculate **Q** from **Q**s which are calculated in 2-4.

Implementation

- Step 2-4
  Some QR decompositions can be calculated in parallel.
  $\Rightarrow$ Batched QR
- Step 5
  Calculate matrix multiplication implicitly using batched matmul.

Batched QR implementation

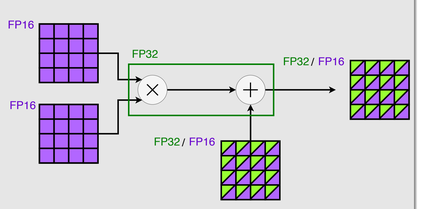
- Parallel QR Factorization for some $m \times n$ matrices ($16 \leq m \leq 32, n \leq 16$)
- Householder QR

Algorithm 2. Householder QR

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

1: $\mathbf{Q}' \Leftarrow \mathbf{I}$, $\mathbf{R} \Leftarrow \mathbf{A}$
2: **for** $i \Leftarrow 0$ **to** $n-1$ **do**
3: $\quad \mathbf{u} \Leftarrow [0 \ \cdots \ 0 \ \mathbf{R}_{i,i} \cdots \mathbf{R}_{m-1,i}]^T$
4: $\quad \mathbf{u}_i \Leftarrow \mathbf{u}_i \pm |\mathbf{u}|$
5: $\quad \mathbf{H} \Leftarrow \mathbf{I} - 2\frac{\mathbf{u}\mathbf{u}^T}{|\mathbf{u}|^2}$
6: $\quad \mathbf{R} \Leftarrow \mathbf{H}\mathbf{R}$
7: $\quad \mathbf{Q}' \Leftarrow \mathbf{H}\mathbf{Q}'$
8: **end for**
9: $\mathbf{Q} \Leftarrow \mathbf{Q}'^T$

TensorCore



- Mixed-precision matrix product and addition circuit

Refinement using TensorCores

$$\mathbf{C}_{FP32} \leftarrow \mathbf{A}_{FP16}\mathbf{B}_{FP16} + \Delta\mathbf{A}_{FP16}\mathbf{B}_{FP16} + \mathbf{A}_{FP16}\Delta\mathbf{B}_{FP16}$$

where

$$\mathbf{M}_{FP16} \xleftarrow{\text{narrowing}} \mathbf{M}_{FP32}$$
$$\Delta\mathbf{M}_{FP16} \xleftarrow{\text{narrowing}} \mathbf{M}_{FP32} - \mathbf{M}_{FP16}$$

TSQR Implementation

Name	Type	TensorCore
FP32-TC	float	Used
FP16-TC	half	Used
FP32-noTC	float	Not used
FP16-noTC	half	Not used

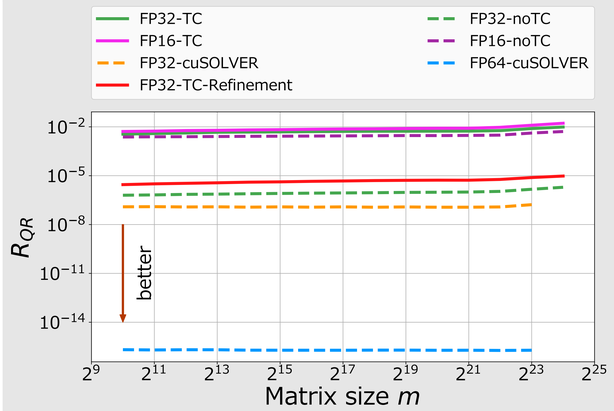
FP32-TC with refinement

- $m \times n$ $(n \leq 16)$

Where to use TensorCore

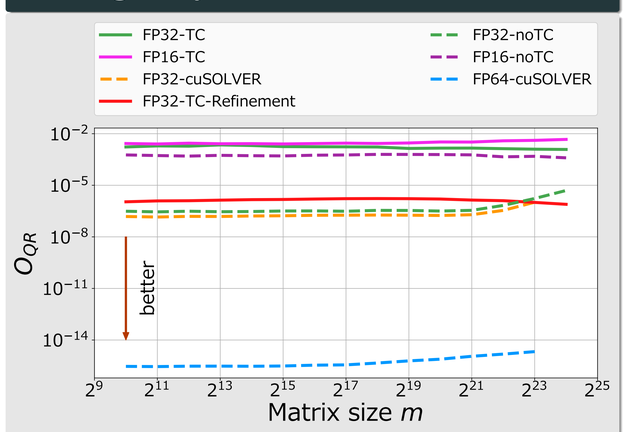
- Calculate **H** (Algo 2. line 5)
- Update **Q**, **R** (Algo 2. line 6, 7)
- Batched Matmul

Residual Evaluation



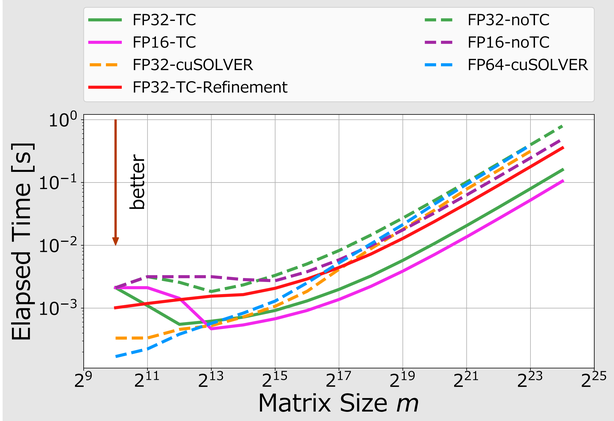
Residual $R_{qr} = \frac{||\mathbf{A}-\mathbf{QR}||_F}{||\mathbf{A}||_F}$

Orthogonality Evaluation

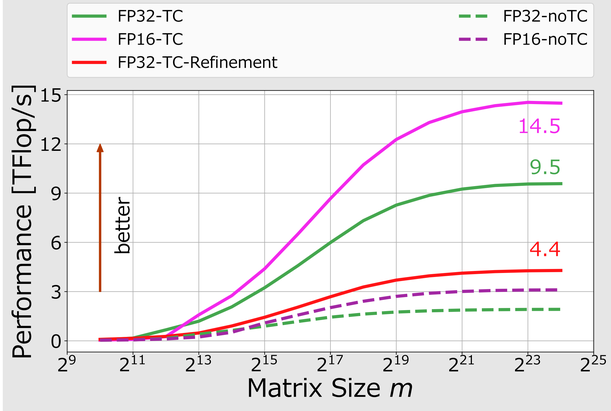


Orthogonality $Q_{qr} = \frac{||\mathbf{I}-\mathbf{Q}^T\mathbf{Q}||_F}{\sqrt{n}}$

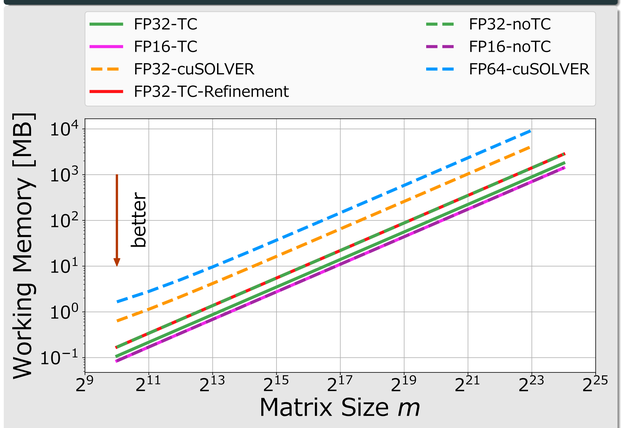
Speed Evaluation



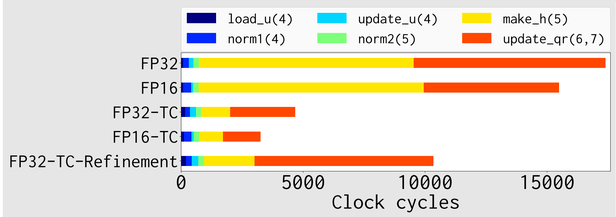
Performance Evaluation



Working Memory Size Evaluation



Computing Time Profile



The number in brackets corresponds to a line number in Algo 2.

Conclusions

- Using TensorCores and refinement, Our approach can calculate TSQR efficiently without much loss of accuracy
- Our approach provides 3.4x faster performance compared to cuSOLVER
- Our approach reduces about 80% of working memory compared to cuSOLVER