High Performance Householder QR Factorization on Emerging GPU Architectures Using Tensor Cores

Yuhan Leng, Gaoyuan Zou, Hansheng Wang, Panruo Wu, and Shaoshuai Zhang

***Abstract*—Since 2017, NVIDIA GPUs have been equipped with specialized units known as Tensor Cores, which demonstrate remarkable efficiency in processing matrix multiplications (GEMMs). Beyond GEMMs, researchers have explored the potential applications of Tensor Cores in matrix factorization, such as QR factorization. However, the inside GEMMs in QR factorization are typically tall and skinny. Compared to compute-bound square GEMMs, these tall and skinny GEMMs are memory bound, leading to suboptimal performance on Tensor Cores. To solve this problem, we indicate the recursive QR factorization can convert the tall and skinny GEMMs to relatively square and large GEMMs, resulting in better performance on Tensor Cores. Besides, we extend the FP16 Tensor-Cores-based QR factorization to accommodate FP32 and FP64 on FP16 and INT8 Tensor Cores, respectively. Additionally, to address the issue of orthogonality loss in the preceding Tensor Cores-based QR factorization, we transition from the Gram-Schmidt to the Householder algorithm while preserving high performance. According to our experimental evaluation conducted on NVIDIA's A100 and GeForce RTX 3090 GPU, the precision levels of FP64, FP32, and FP16 are up to 6.22x, 8.67x, and 4.03x faster, respectively, than the current state-of-the-art implementations.**

***Index Terms*—HPC, GPGPU, numerical linear algebra, tensor cores, mixed-precision algorithms.**

I. Introduction

THE GROWING demand for training large neural networks has steered the modern GPU design towards low-precision matrix engines. A noteworthy instance of such low-precision matrix engines is Nvidia's Tensor Cores, which outpaces FP32 (single precision) SIMT cores significantly. The evolution of GPUs has been rapid in recent years, elevating the peak FP16 Tensor Cores' performance from 112 TFLOPs to around 1 PFLOPs (refer to Table I for details). Furthermore, on the latest A100 and H100 GPUs, FP64 (double precision) GEMM can also be executed on Tensor Cores, delivering comparable speed to SGEMM on FP32 SIMT cores.

TABLE I
THE GEMM PERFORMANCE (IN TFLOPS) OF DIFFERENT GENERATIONS OF NVIDIA GPUS:

GPU \ Routine	DGEMM	SGEMM	TC-GEMM
P100	4.7	9.3	-
V100	7.0	14.0	112.0
A100	19.5	19.5	312.0
H100	67.0	67.0	989.0

In recent times, the utility of Tensor Cores has been extended beyond deep neural network training/inference. Fundamental numerical linear algebra operations, such as LU factorization [1], QR factorization [2], and eigenvalue decomposition [3], can also benefit from Tensor Cores. However, traditional algorithmic designs for numerical linear algebra struggle to harness a substantial portion of the peak performance from Tensor Cores due to their exceptionally fast computations, turning data movement into a bottleneck for conventional tiling-based algorithms. To address this challenge, Tensor-Cores-based QR factorization [2] combines tall and skinny QR factorization ([4]) and recursive Gram-Schmidt QR factorization to achieve superior performance on Tensor Cores. It also introduces the use of the conjugate gradient descent method for solving linear least square problems (CGLS) to handle accuracy loss issues stemming from FP16 Tensor Cores.

Nevertheless, there are several drawbacks to the mentioned Tensor-Cores-based QR factorization. The Gram-Schmidt QR factorization often induces a loss of orthogonality, and while the re-orthogonalization technique can partially restore orthogonality, it requires twice the computations. Additionally, the re-orthogonalization process falters when the matrix is extremely ill-conditioned. Furthermore, in the context of linear least squares problems involving matrices with high condition numbers, employing low-precision QR factorization as a preconditioner within the Conjugate Gradient Least Squares (CGLS) method generally requires an increased number of iterations. In some cases, this approach may even fail to achieve convergence.

To overcome these challenges, we first replace the Gram-Schmidt QR factorization with Householder QR factorization to preserve orthogonality. Second, we extend the precision of QR factorization from FP16 to FP32 and FP64 to handle more complex tasks.

Thus, the contributions of this paper can be summarized as follows:

Received 1 March 2024; revised 12 December 2024; accepted 17 December 2024. Date of publication 25 December 2024; date of current version 20 January 2025. This work was supported in part by the University of Electronic Science and Technology of China startup under Grant A1098531023601465 and in part by NSF CAREER under Award 2146509. Recommended for acceptance by B. Ucar. *(Corresponding author: Shaoshuai Zhang.)*

Yuhan Leng, Gaoyuan Zou, Hansheng Wang, and Shaoshuai Zhang are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China (e-mail: 202221080503@std.uestc.edu.cn; 202322080434@std.uestc.edu.cn; wanghansheng@std.uestc.edu.cn; szhang94@uestc.edu.cn).

Panruo Wu is with the Department of Computer Science, Univeristy of Houston, Houston, TX 77004 USA (e-mail: pwu7@uh.edu).

Digital Object Identifier 10.1109/TPDS.2024.3522776

- We conduct a thorough analysis and evaluation of the performance bottleneck of Householder QR factorization on Tensor Cores.
- We implement and optimize Householder QR factorization with FP64, FP32, and FP16 precision on different types of GPUs, demonstrating superior performance compared to state-of-the-art QR factorization in the cuSOLVER library.
- We demonstrate that compared to the FP16 Gram-Schmidt QR factorization, the proposed FP16 Householder QR factorization is more stable in solving linear least square problems.

The remainder of this paper is organized as follows: Section II delves into related work concerning Tensor Cores technology and QR factorization. Section III furnishes background knowledge on Householder QR factorization and its recursive formulation. In Section IV, we dissect the performance bottleneck of Householder QR factorization, while Section V introduces our implementation and optimization techniques. Section VI presents the experimental results of QR factorization and its applications. Section VII concludes the paper and outlines our future work.

II. Related Work

A. Tensor Core Technology

Tensor Core technology was introduced by NVIDIA in 2017 on its Tesla V100 GPU [5] with the aim of accelerating mixed-precision GEMMs, thereby improving the efficiency of the training/inference process in deep neural networks. Presently, Tensor Cores have been incorporated into NVIDIA's Turing, Ampere, and Hopper architectures. In the second generation of Tensor Cores, NVIDIA introduced INT8 on its Turing architecture, significantly boosting inference throughput and yielding substantial efficiency increments [6]. On the latest Hopper Tensor Cores, its INT8 Tensor Cores exhibit a performance 6x faster than FP16 Tensor Cores on Ampere [7], [8]. Early research on Tensor Cores [9], [10], [11] includes performance modeling, architectural exploration, and implementations.

Initially, Tensor Cores found common application in training and inference for deep neural networks. For instance, Zhu et al. [12] proposed new instructions and microarchitectural optimizations on Tensor Cores to enhance their suitability for accelerating sparse neural networks. Furthermore, researchers have introduced a computational framework for quantized graph neural networks based on Tensor Cores [13], significantly improving inference efficiency. Additionally, Feng [14] introduced a neural network framework capable of arbitrary precision, combined with Tensor Cores, resulting in substantial acceleration.

Recently, researchers have extended the usage of Tensor Cores beyond deep learning to other fields. Numerical linear algebra, in particular, stands to benefit significantly, as its internal GEMMs can be directly replaced by GEMMs on Tensor Cores. Utilizing Tensor Cores in LU factorization [1], [15] and QR factorization [2] has led to a significant speedup at the cost of some accuracy loss. Researchers have adapted iterative methods, including GMRES and CGLS, to restore accuracy to high precision. Other types of computations using Tensor Cores include FFT [16], reduction and scan [17], stencil computations [18], and support vector machines using low-rank approximation [19]. Notably, in terms of recovering accuracy loss from low-precision Tensor Cores, Ootomo implemented SGEMM [20] and DGEMM [21] on FP16 Tensor Cores and INT8 Tensor Cores, respectively. The Tensor-Cores-based SGEMMs (TC-SGEMMs) and DGEMMs (TC-DGEMMs) exhibit the same accuracy as conventional SGEMMs and DGEMMs while surpassing the theoretical FP32/64 peak performance. TC-SGEMM was adopted in eigenvalue decomposition [22], demonstrating superior performance compared to MAGMA [23] and cuSOLVER.

B. QR Factorization

The QR factorization holds a pivotal role in data analysis and scientific engineering, addressing tasks such as orthogonalization, low-rank approximation [24], and two stage eigenvalue/singular value decomposition [25], [26]. For instance, in QR-based Singular Value Decomposition (QR-SVD), QR factorization helps obtain orthogonal matrices and upper triangular matrices, improving the efficiency of computing the singular value decomposition on a tall and skinny matrix [2].Various classical QR factorization algorithms exist, each with its own characteristics. Householder QR factorization, deemed the most stable method, utilizes Householder reflection to derive the orthogonal basis. Classic/modified Gram-Schmidt QR factorization, while less stable, exhibits improved parallelism and lower time complexity. QR factorization using Givens rotation achieves excellent orthogonality, particularly targeting sparse matrices. However, traditional QR factorization methods face challenges in achieving optimal performance when dealing with tall and skinny matrices, ill-conditioned matrices, and singular matrices. Moreover, they encounter limitations in maintaining matrix orthogonality and exposing locality, prompting extensive research efforts to address these issues.

In-depth investigations into the QR factorization of tall and skinny matrices have been conducted [27], exploring critical paths and algorithmic performance. The Cholesky QR method's advantages in terms of orthogonalization and numerical stability are discussed in [28], where algorithmic performance is further enhanced through the incorporation of mixed precision. Building on Cholesky QR, [29] extends the shifted Cholesky QR3 algorithm by introducing shifts into the computation of the Gram matrix, resulting in an algorithm with superior numerical stability and parallelism, particularly effective for matrices with large condition numbers. Additionally, [30] proposes a solution to the performance limitation in Householder QR factorization with column pivoting by selecting axis vectors through random projection, achieving significant acceleration. Researchers have explored avenues to overcome the performance bottlenecks of QR decomposition from other perspectives, as demonstrated in the communication-avoiding rank-revealing QR factorization algorithm called CARRQR [31], which reduces data transmission for improved speed.

With the advent of GPUs, substantial research efforts have been dedicated to leveraging GPU parallelism for accelerated factorization calculations, surpassing the performance of standard numerical computation libraries like LAPACK [32] and Intel MKL. Evaluations of five different LU factorization and QR factorization implementations on CPU and GPU [33] are conducted, yielding in-depth analyses. To enhance GPU parallelism utilization, GPU-based tall and skinny QR factorization is proposed on CUDA cores [4] and Tensor Cores [34]. MAGMA's [23] QR factorization routine employs a hybrid algorithm performing panel factorization on CPU while updating trailing matrices on GPU, demonstrating competitive performance relative to cuSOLVER,[1] a likely state-of-the-art implementation on GPUs for general matrices.

III. Background

QR factorization is one of the most vital matrix factorizations in numerical linear algebra. Its derived applications include Linear Least Squares (LLS) Solver and Low-Rank Approximation (LRA), and these applications play important roles in real-world problems. QR factorization aims to factorize a matrix to a product of an orthogonal matrix Q and an upper triangular matrix R, and it's almost always an important building block of any numerical linear algebra packages such as LAPACK, ScaLAPACK. On GPU, Nvidia provides a well-optimized linear algebra library named cuSOLVER,[2] containing LU, QR, SVD and EVD. MAGMA is a more comprehensive tool that works on hybrid CPU/GPU architecture. The most performant and stable algorithm (used in all the above-mentioned packages across different architectures: distributed memory, multi-core, GPUs, and single-core CPU) is the tiled Householder QR factorization. We will typically use `D/SGEQRF` to denote such implementation in double and single precision Generalized QR Factorization from cuSOLVER.

Mathematically, QR factorization (thin version) is a matrix decomposition technique that decomposes a matrix A into a product of an orthonormal matrix Q ($Q^TQ = I$) and an upper triangular square matrix R:

$$\underbrace{A}_{m\times n, m\geq n} = \underbrace{Q}_{m\times n, \text{orthonormal}} \times \underbrace{R}_{n\times n, \text{upper triangular}}$$

In general, there are three basic algorithms to perform QR factorization: Gram-Schmidt process, Householder reflection, and Givens rotation. Among them, Givens rotation is normally used for sparse matrices or matrices with special structure, so we are only focusing on Gram-Schmidt and Householder reflection as the base background of QR factorization in this paper.

A. Conventional QR Factorization Algorithms

1) Gram-Schmidt QR Factorization: The Gram-Schmidt process aims to find a set of orthonormal vectors in an inner product space. Given an array of linear independent vectors $[a_1|a_2|a_3|\ldots|a_n]$, the Gram-Schmidt will find the orthonormal basis one by one. Then the set of orthonormal vectors $[q_1|q_2|q_3|\ldots|q_n]$ is given by (1).

$$\begin{aligned}
u_1 &= a_1, & q_1 &= u_1/||u_1|| \\
u_2 &= a_2 - \text{Proj}_{u_1}(a_2), & q_2 &= u_2/||u_2|| \\
u_3 &= a_3 - \text{Proj}_{u_1}(a_3) - \text{Proj}_{u_2}(a_3), & q_3 &= u_3/||u_3|| \\
&\vdots & &\vdots \\
u_n &= a_n - \sum_{j=1}^{n-1} \text{Proj}_{u_j}(a_n), & q_n &= u_n/||u_n||
\end{aligned} \tag{1}$$

in which $\text{Proj}_u(a) = uu^Ta = u(u^Ta)$ is the orthogonal projection of vector a onto unit vector u. We also obtain R during the Gram-Schmidt process: $r_{ij} = q_i^Ta_i$.

Depending on the evaluation order of the procedure described in (1), there are two mathematically equivalent but numerically different variants called classic Gram-Schmidt (CGS) and modified Gram-Schmidt (MGS). CGS executes row by row in (1), whereas MGS subtracts $\text{Proj}_{u_i}(a_j)$ from a_j for all $j > i$ as soon as u_i is computed. MGS can be visualized as evaluating (1) with row and column interleaved. This subtle difference has important implications in numerical stability and parallelism exposed. MGS is more stable but less parallel [35]. The block-based computation in MGS is less efficient, requiring additional communication overhead, and fails to fully exploit the computing capacity of modern hardware architectures (e.g., the excessive use of BLAS2 operations). Recently, several variants of the Modified Gram-Schmidt (MGS) algorithm have been proposed to enhance parallelism and achieve performance similar to Classical Gram-Schmidt (CGS). For example, Left-Looking MGS improves parallelism at the cost of stability, while MGS2 and MGS+ increase parallelism by adding extra floating point operations [36].

The Gram-Schmidt process, as presented above, is inefficient on a hierarchical memory system due to the low data locality of vector-matrix operations. To enhance locality, blocking must be employed, allowing the orthogonal projection of multiple vectors onto multiple orthonormal vectors to be performed in one shot. Classical Gram-Schmidt (CGS) can be trivially blocked, as it can be directly "upgraded" into a blocking algorithm by considering the vectors a_i, u_i not as individual vectors but as a block matrix (a group of column vectors). Modified Gram-Schmidt (MGS), on the other hand, is not straightforward to block.

2) Householder QR Factorization: The Householder reflector is an orthogonal projection constructed from reflection against a hyperplane. It is useful to orthogonally transform a given vector to an axis (thereby eliminating all components but one). Specifically, given a vector x, the orthogonal matrix $H(v) = I - 2vv^T/(v^Tv)$ where $v = ||x||e_1 - x$ will map x to the first axis: $H(v)x = [||x||, 0, \ldots, 0]^T$. Householder QR algorithm orthogonally eliminates the below-diagonal elements

[1] https://docs.nvidia.com/cuda/cusolver/index.html
[2] https://docs.nvidia.com/cuda/cusolver/

TABLE II
CANONICAL COMPLEXITY AND STABILITY COMPARISON WITH CGS, MGS AND HOUSEHOLDER

Algorithm	CGS	MGS	Householder
FLOPS	$2mn^2$	$2mn^2$	$2mn^2 - \frac{2}{3}n^3$
Backward Error Bound	$O(\epsilon)$	$O(\epsilon)$	$O(\epsilon)$
Orthogonality Error Bound	$O(\epsilon\kappa^{n-1}(A))$	$O(\epsilon\kappa(A))$	$O(\epsilon)$
Backward Error Bound	$O(\epsilon)$	$O(\epsilon)$	$O(\epsilon)$
Reorthogonalization	$4mn^2$	$4mn^2$	
Generating Explicit Q			$4mn^2 - \frac{4}{3}n^3$

Complexity is #flops for unblocked versions.

of A column by column, by successively pre-multiplying Householder reflections; see (2).

$$H_1A = \begin{bmatrix} r_{11} & * & \cdots & * \\ 0 & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & * & \cdots & * \end{bmatrix}, H_2H_1A = \begin{bmatrix} r_{11} & r_{12} & \cdots & * \\ 0 & r_{22} & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & * \end{bmatrix}$$

$$\ldots, \quad H_n \cdots H_2H_1A = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix} = R \tag{2}$$

At the final stage, we have the QR factorization with orthogonal $Q = H_1^T H_2^T \cdots H_n^T$.

In practice, pre-multiplying a Householder matrix with a matrix $H \times A$ is inefficient on hierarchical memory system due to low data locality, since $H \times A = (I - 2vv^T)A = A - 2v(v^TA)$ involves only vector-matrix operations. To improve data locality, a technique called blocking must be used to group several Householder reflectors into a single matrix application: $H_kH_{k-1}\ldots H_1 = I - WY^T$, where $W, Y \in \mathbb{R}^{n\times k}$; ref WY representation in [22]. This is critical in achieving high performance of Householder QR.

3) Comparison: Table II illustrates the distinctions (in terms of complexity and stability) between the canonical CGS, MGS, and Householder QR factorization. It is worth noting that while some references [37] point out that the orthogonality error of CGS is typically constrained by the square of the condition number, this is not contradictory to other references. The orthogonality error of CGS generally only worsens to $O(\epsilon\kappa^{(n-1)}(A))$ in cases of high matrix dimensions or poor numerical conditioning [36], [38], [39]. To address this issue, common variants like CGS-P leverage geometric relationships between vectors, which help mitigate floating-point calculation errors and improve numerical stability to $O(\epsilon\kappa^2(A))$. In summary, the differences among these QR algorithms are as follows: Householder QR is stable, but its Q factor is implicit, as computing the exact Q incurs additional computational cost. MGS is more stable than CGS, but requires more communication when performing block-wise computation.

Due to roundoff errors, the numerical system does not compute exact QR factorization. To measure the accuracy of the factorization results, two errors are considered. The first one is the backward error, denoted as $\frac{||A-\hat{Q}\hat{R}||}{||A||}$, where A is the matrix to be factorized and $\hat{Q}$ and $\hat{R}$ are the factorization results. CGS, MGS, and Householder QR factorization exhibit the same backward error bound, which is the machine ϵ. Another measurement is orthogonality, which gauges how orthogonal $\hat{Q}$ is and is denoted as $||I - \hat{Q}^T\hat{Q}||$. The orthogonality error bound for the original versions of CGS and MGS is worse than that for Householder, implying that CGS and MGS methods may struggle with real ill-conditioned systems. Fortunately, with continued improvements to the Gram-Schmidt algorithms, according to [36], certain variants can achieve the same level of accuracy as Householder QR under specific conditions, or attain comparable stability through reorthogonalization. A detailed analysis of the errors will be presented in the experiments section.

B. Recursive Formulation

The state-of-the-art implementations in LAPACK [32] and MAGMA [23] consistently employ tiling Householder QR factorization (WY representation). This preference arises because Householder QR factorization is more cost-effective than Gram-Schmidt QR factorization when the orthogonal matrix Q is not needed. Additionally, Householder reflection ensures orthogonality during the factorization process. However, in the context of using Tensor Cores, especially FP16 Tensor Cores, the tiling algorithm may not fully harness the power of Tensor Cores, due to its application in dimension-limited tall and skinny matrices, where the arithmetic intensity is constrained. A detailed performance analysis and data will be presented in Section IV. Consequently, recent research [2], [3] explores methods not typically considered on modern computer architectures, such as recursive QR factorization and WY-based Eigenvalue decomposition. These methods provide reasonable shapes of General Matrix-Matrix Multiplications (GEMMs) for acceleration on Tensor Cores, albeit at the cost of increased computational complexity.

In terms of QR factorization, the process of recursive Gram-Schmidt QR factorization is assembled as follows:

$$[A_1|A_2] = [Q_1|Q_2]\left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right] \tag{3}$$

Given a matrix A, we divide evenly its columns into two halves, denoted by $A = [A_1|A_2]$. We first QR factorize the first half $A_1 = Q_1R_{11}$, and then compute north-east quarter $R_{12} = Q_1^TA_2$. Next we update the second half $A_2 = A_2 - Q_1R_{12}$. Finally, QR factorize the updated second half $A_2 = Q_2R_{22}$. Note that the QR factorization of the two halves can be further divided into another two recursions. There is no additional computation introduced during the recursion process.

The recursive formulation becomes more intricate when it comes to Householder QR factorization, as the WY representation has to be reformed during the recursion, thereby introducing additional computations:

$$A_1 = Q_1 \times R_1 = (I - W_1 \times Y_1^T) \times R_{11}$$

$$A_2 = A_2 - Y_1 \times W_1^T \times A_2$$

$$
\begin{aligned}
A_2 &= \left[\frac{R_{12}}{A_2}\right] \\
A_2 &= Q_2 \times R_2 = (I - W_2 \times Y_2^T) \times R_{22} \\
W_2 &= \left[\frac{0}{W_2}\right] \\
Y_2 &= \left[\frac{0}{Y_2}\right] \\
W &= [W_1 \,|\, -W_1 \times Y_1^T \times W_2 + W_2] \\
Y &= [Y_1 \,|\, Y_2] \\
R &= \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & R_{22} \end{array}\right]
\end{aligned} \tag{4}
$$

Note that in the recursive formulation shown in (4), the entire W is not necessary to be formed if Q is not needed. However, W_1 generated from factorizing A_1 still have to be fully formed before updating A_2, which leads to around 1.5x the number of mathematical operations compared to the tiling Householder algorithm.

C. Tall and Skinny QR Factorization

The conventional panel factorization involves sequentially dependent iterations, and the working set is the entire panel, which cannot fit in fast memory on GPU (register files + shared memory). Fortunately, for QR, there's a tall and skinny QR (TSQR) [4] variant that simultaneously improves parallelism and data locality, at the cost of more computations. The idea of TSQR can be illustrated in (5).

$$
\begin{aligned}
\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}
&\overset{①}{=}
\begin{bmatrix} Q_{11}R_1 \\ Q_{12}R_2 \\ Q_{13}R_3 \\ Q_{14}R_4 \end{bmatrix}
\overset{②}{=}
\begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix}
\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \\
&\overset{③}{=}
\begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix}
\begin{bmatrix} Q_{21} \\ Q_{22} \\ Q_{23} \\ Q_{24} \end{bmatrix} R \\
&\overset{④}{=}
\begin{bmatrix} Q_{11}Q_{21} \\ Q_{12}Q_{22} \\ Q_{13}Q_{23} \\ Q_{14}Q_{24} \end{bmatrix} R
\overset{⑤}{=} QR
\end{aligned} \tag{5}
$$

In (5), there are five steps indicated by the numbers over the equality sign. In step , we divide a tall matrix A evenly into four smaller matrices (still tall, more rows than columns), and QR factorize them independently. In step , we stack the R factors vertically. Note that the number of rows of the R factors is less than the number of rows of the original A. In step , we factorize the vertically stacked Rs (potentially carrying out this process recursively). In , we perform four matrix-matrix multiplications for the four corresponding Q factors. In , we reinterpret the result as the QR factors of the original A. The reason Q is orthogonal is that in step , the four matrix-matrix multiplications are equivalent to the product of two orthogonal matrices (second line) and, therefore, is orthogonal [2].

D. SGEMM and DGEMM on Tensor Cores

According to the research by Ootomo et al. [20], [21], it is possible to perform SGEMMs and DGEMMs on Tensor Cores without any accuracy loss. Since the technology of TC-SGEMMs and TC-DGEMMs is not the same, we will introduce the background separately.

1) SGEMMs on FP16 Tensor Cores: The accumulation process in matrix multiplications on FP16 Tensor Cores can be executed using FP32 precision. This feature allows us to recover the accuracy loss by performing two additional TC-GEMMs:

$$
\begin{aligned}
A_{FP32} \times B_{FP32} = &\ (A_{FP16} + \Delta A_{FP16}) \\
&\times (B_{FP16} + \Delta B_{FP16})
\end{aligned}
$$

Compared to the original single FP32 matrix multiplication, this formula requires performing four FP16 multiplications and three FP32 additions. Note that the TC-GEMM $\Delta A_{FP16} \times \Delta B_{FP16}$ can be bypassed as it's trivial for the final results.

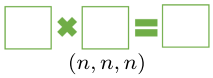
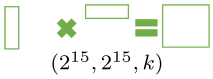
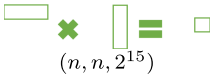
Theoretically, with the above equation, the accuracy loss can be recovered. However, based on experimental results [20], [40], this method cannot work when the matrix size, especially when k, is large. Ootomo et al. [20] investigate the error behavior in-depth and figure out that the accumulations inside Tensor Cores are rounding to zeros instead of rounding to the nearest. Therefore, when k is large, the rounding errors are accumulated and cannot be recovered at all.

To solve the error from rounding to zeros, the authors design their own `wmma` kernel[3] and change the rounding behavior inside Tensor Cores to nearest rounding. Finally, the accuracy loss is recovered using FP16 Tensor Cores. On the A100 GPU, the peak FP16 Tensor Core performance can reach 312 TFLOPS, while the FP32 peak performance is 19.5 TFLOPS. Therefore, even though our implementation introduces extra computational and communication overhead for storing and reading FP16 matrices, the achieved FP32 matrix multiplication performance still exceeds the theoretical FP32 peak performance on the A100 GPU by over 2x.

2) DGEMMs on INT8 Tensor Cores: Different from the TC-SGEMMs, the FP16 Tensor Cores don't support accumulations with FP64 precision; otherwise, the aforementioned method could also work on FP64 precision. As a result, Ootomo et al. [21] abandon the above method and use INT8 precision to simulate FP64 numbers by splitting FP64 numbers into several blocks. Note that as the numbers may not occupy all 64 bits, the authors search the range of the numbers and calculate the minimal length of INT8 arrays to reduce the number of arithmetic operations. Experimentally, on emerging GPUs such as RTX

[3]The term "wmma kernel" refers to a specialized CUDA kernel utilizing the Warp Matrix Multiply-Accumulate (WMMA) operations to accelerate matrix multiplications using Tensor Cores. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

TABLE III
PERFORMANCE COMPARISON (IN TFLOPS) BETWEEN TC-GEMMS, TC-SGEMMS AND SGEMMS ON A100 GPU, DGEMMS AND TC-DGEMMS ON RTX 4090 GPU WITH THREE TYPES OF GEMM SHAPES AND SIZES

GEMM Shape	Routine	32	64	128	256	512	1024	2048	4096	8192	16384	32768
 (n, n, n)	TC-GEMM	0.005	0.028	0.293	2.048	13.11	58.25	111.8	216.1	259.8	238.3	187.3
	TC-SGEMM	0.005	0.030	0.273	2.340	7.085	10.87	32.58	40.68	44.05	39.29	38.74
	SGEMM	0.006	0.032	0.315	1.638	6.899	14.79	17.57	19.16	19.06	19.12	19.15
	DGEMM	0.0002	0.001	0.012	0.082	0.453	0.917	1.093	1.240	1.241	1.242	-
	TC-DGEMM	0.0002	0.001	0.010	0.073	0.411	0.560	2.120	4.374	7.042	9.109	-
 $(2^{15}, 2^{15}, k)$	TC-GEMM	27.20	52.55	89.93	161.3	234.6	259.6	217.9	251.6	274.4	269.1	187.3
	TC-SGEMM	17.01	25.65	31.98	34.78	38.44	40.83	41.86	39.09	38.77	37.93	38.74
	SGEMM	17.21	18.24	18.71	19.00	18.98	18.23	18.28	19.12	19.15	19.08	19.15
	DGEMM	1.295	1.169	1.170	1.187	1.226	1.235	1.238	1.241	1.242	1.242	-
	TC-DGEMM	1.291	1.169	1.205	1.266	1.235	1.412	2.987	-	-	-	-
 $(n, n, 2^{15})$	TC-GEMM	0.736	2.759	8.525	8.648	53.77	105.5	163.8	231.1	263.2	284.5	187.3
	TC-SGEMM	2.185	3.318	7.944	11.68	13.71	12.78	34.34	43.60	43.56	38.91	38.74
	SGEMM	0.006	0.032	0.315	1.638	6.899	14.79	17.57	19.16	19.06	19.12	19.15
	DGEMM	0.017	0.062	0.260	0.795	1.102	1.122	1.245	1.235	1.228	1.241	-
	TC-DGEMM	0.016	0.071	0.271	0.777	1.093	3.561	4.810	9.539	9.839	9.633	-

The dimensions (*m,N, k*) indicate that a matrix of size $m \times k$ is multiplied by a matrix of size $k \times n$, resulting in a matrix of size $m \times n$.

4090 and RTX 3090, whose FP64 precision computing capacity is limited, the proposed TC-DGEMMs can reach around 7x speedup compared to theoretical FP64 peak performance.

E. The Proposed Householder QR Algorithm Workflow

Building upon the improvements and the identified strengths and weaknesses of the existing methods discussed above, this section outlines the workflow of the proposed Householder QR optimization approach. The underlying optimization principles will be comprehensively detailed in the following sections on performance analysis and implementation.

Depending on the computing capacity of the hardware, we propose two algorithms (tiling/recursive) for different hardware and precisions. For GEMMs that benefit from Tensor Cores for improved performance, the recursive algorithms are preferred. This choice is based on the sensitivity of TC-GEMMs and TC-S/DGEMMs to matrix shapes and sizes, as indicated in Table III. For GEMMs that do not utilize Tensor Cores, the tiling algorithm is considered to enhance parallelism. Algorithm 1 illustrates the workflow of the proposed QR factorization algorithm, with the tiling algorithm presented in a recursive manner for better comparison to the recursive approach. It is noteworthy that in the tiling method, the calculation of W (line 16) is necessary only if Q is needed. Additionally, generating Q requires extra GEMMs, which may result in significant additional time overhead.

The Algorithm 1 makes it clear that the selection between the two variants depends on whether TC-GEMMs, TC-SGEMMs, and TC-DGEMMs are involved. If these Tensor Core operations are not required, meaning that the GEMM shapes and sizes do not significantly impact performance, the tiling variant will be chosen, and vice versa. From a hardware perspective, for instance, on A100 and H100 GPUs, the FP16 and FP32 precisions will use the recursive variant, while FP64 precision QR factorization will use the tiling variant, as TC-DGEMMs perform worse than primal DGEMMs on A100. However, for other new generation GPUs like RTX 3090, RTX 4090 and A6000, the FP16 and FP64 precisions will adopt the recursive variant to obtain better inside GEMM shapes and sizes.

IV. PERFORMANCE ANALYSIS

A. Trailing Matrix Update Performance

Table III displays a performance comparison between DGEMMs, DGEMMs using INT8 Tensor Cores (TC-DGEMMs), SGEMMs, SGEMMs using Tensor Cores (TC-SGEMMs), and Tensor-Cores-based GEMMs (TC-GEMMs). It is evident from the table that the TC-GEMMs, TC-SGEMMs and TC-DGEMMs require larger matrix sizes to reach peak performance, indicating that the performance of GEMMs on Tensor Cores is more sensitive to matrix shapes than SGEMMs.

The performance discrepancies between normal GEMMs and GEMMs on Tensor Cores arise from the exceptionally fast calculations facilitated by Tensor Cores. As the input and output are FP32, the data movements between registers, shared memory, and device memory remain the same. However, when calculations are extremely fast, data movements can become the bottleneck. This scenario is explicated by the roofline model [41]:

$$I = \frac{W}{Q},$$

where W is the number of arithmetical operations (expressed as FLOP) and Q is the memory traffic (expressed as bytes). For a given GEMM size m, n, k, the arithmetic intensity (I) of square GEMMs ($m = n = k$) is $\frac{2}{3}n$, and the arithmetic intensity of matrix outer product ($m = n$, $n \gg$k) is $\frac{2nk}{n+2k}$. The arithmetic intensity of the matrix inner products ($m = n$, $n \ll k$) is the same as that of matrix outer products, but with a much smaller n compared to k. It is evident that square GEMMs, given the same n, generally have the highest arithmetic intensity, while inner products typically have the lowest due to their much smaller n compared to k.

More specifically, for QR factorization, the main differences between conventional tiling algorithm and recursive algorithm are the GEMM shapes. For the tiling algorithms, the

Algorithm 1: The Workflow of Proposed QR Factorization on Different Conditions.

function QR A ▷Return W, Y, R s.t. $A = (I - WY^T)R$
  A is $m \times n$ matrix
  if $n \leq nb$ **then**
    return PanelQR(A)
  end if
  Divide the columns of $Q = I - WY^T$ into two partitions, and A, R into quadrants
  if Tensor Cores cannot provide acceleration (when $m \gg nb$) **then**
    $\overbrace{\begin{bmatrix} A_{11} \\ \hline A_{21} \end{bmatrix}}^{n_1} \mid \overbrace{\begin{bmatrix} cA_{12} \\ \hline A_{22} \end{bmatrix}}^{n-n_1}] = [\overbrace{Q_1}^{nb} \mid \overbrace{Q_2}^{n-nb}] \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right]$
    $[W_1, Y_1, R_{11}] = \mathbf{QR}\left(\begin{bmatrix} A_{11} \\ \hline A_{21} \end{bmatrix}\right)$
    $\begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix} \leftarrow Q_1 \begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix} = (I - W_1 Y_1^T) \begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix}$
    $R_{12} = A_{12}$
    $[W_2, Y_2, R_{22}] = \mathbf{QR}\left(\begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix}\right)$
    $W_2 = -W_1 Y_1^T W_2 + W_2$
    return $[W_1|W_2], [Y_1|Y_2], \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right]$
  else
    $\overbrace{\begin{bmatrix} A_{11} \\ \hline A_{21} \end{bmatrix}}^{\frac{n}{2}} \mid \overbrace{\begin{bmatrix} cA_{12} \\ \hline A_{22} \end{bmatrix}}^{\frac{n}{2}} = [\overbrace{Q_1}^{\frac{n}{2}} \mid \overbrace{Q_2}^{\frac{n}{2}}] \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right]$
    $[W_1, Y_1, R_{11}] = \mathbf{QR}\left(\begin{bmatrix} A_{11} \\ \hline A_{21} \end{bmatrix}\right)$
    $\begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix} \leftarrow Q_1 \begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix} = (I - W_1 Y_1^T) \begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix}$
    $R_{12} = A_{12}$
    $[W_2, Y_2, R_{22}] = \mathbf{QR}\left(\begin{bmatrix} A_{12} \\ \hline A_{22} \end{bmatrix}\right)$
    $W_2 = -W_1 Y_1^T W_2 + W_2$
    return $[W_1|W_2], [Y_1|Y_2], \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right]$
  end if
end Function

GEMMs typically involve tall and skinny matrices, and these GEMMs cannot fully extract the peak performance of FP16 Tensor Cores. In contrast, the recursive formulation provides relatively square GEMMs, which can potentially improve the utilization efficiency of Tensor Cores. Fig. 1 shows the inside GEMMs performance in Householder QR factorization (matrix size 32768×32768) using tiling/recursive variant on RTX 4090 GPU. These GEMM operations serve the overall QR decomposition process. In employing a tiling variant for GEMM operations, the matrices are typically tall and skinny, with the number of columns fixed to a certain size (as illustrated in Fig. 1, where we selected 128, a commonly-used tile size). Consequently, the data intensity of the GEMMs is insufficient to fully utilize the

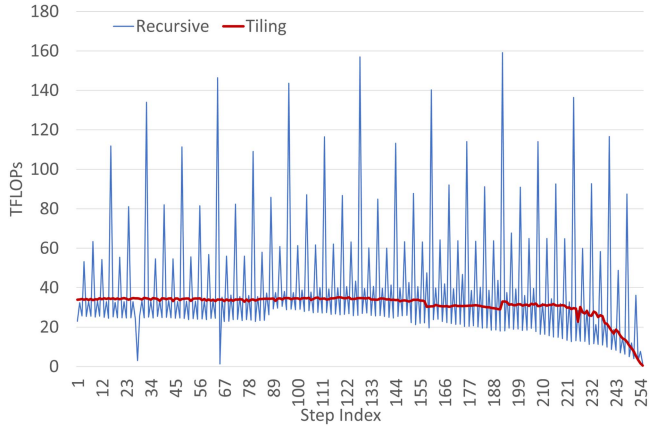


Fig. 1. The inside TC-GEMMs performance comparison between recursive and tiling Householder QR factorization.

peak performance of the Tensor Cores. The performance of the GEMMs, indicated by the red line in Fig. 1, consistently remains around 40 TFLOPs during the iteration steps, compared to the peak performance of the RTX 4090, which is approximately 160 TFLOPs.

In contrast, the aforementioned dimension in recursive variant is not fixed and it depends on the recursion depth. The blue line displays GEMMs performance in recursive Householder QR factorization, we can observe that the performance fluctuates dramatically along the recursion steps because the GEMM shapes change rapidly in the recursive variant. However, we can find in most of the steps, the TFLOPs in recursive variant is lower than in tiling variant. Fortunately, these GEMMs typically have much smaller size than those GEMMs which have much higher TFLOPs. Quantitatively, in terms of the number of mathematical operations, only less than 10% of the computations are executed slowly, while the more expensive GEMMs can be computed over 100 TFLOPs, leading to a better utilization of Tensor Cores in Householder QR factorization.

B. Panel Factorization Performance

Unlike the trailing matrix update, panel factorization (QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$, where $m \gg n$) is typically memory-bound rather than compute-bound. Based on the experimental testing using MAGMA `dgeqrf` routine, although panel factorization generally accounts for less than 10% of the total FLOPS, its time cost can constitute over 30% of the total elapsed time. Regrettably, current state-of-the-art implementations often treat panel factorization as a standard QR factorization, resulting in considerable overhead. Fortunately, tall and skinny QR factorization demonstrates the potential to significantly expedite the panel factorization process. Fig. 2 illustrates the elapsed time for QR factorizing a matrix with dimensions $m \times 32$ on a RTX 4090 GPU. It is evident that for sufficiently large values of m, tall and skinny QR factorization outperforms the cuSOLVER `SGEQRF` routine by approximately a factor of 10x.

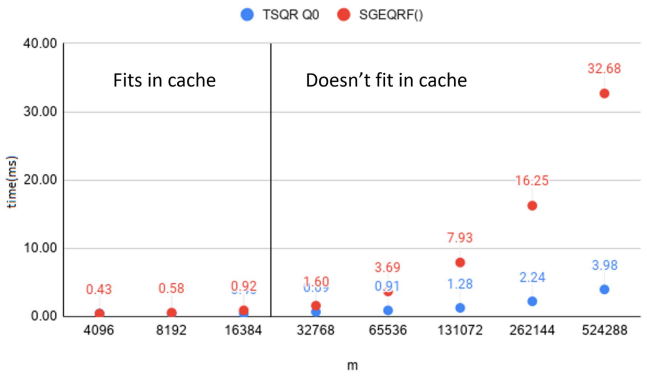


Fig. 2. $m \times 32$ tall-skinny matrix factorization: recursive TSQR QR0 vs. cuSOLVER SGEQRF on RTX 4090 GPU.

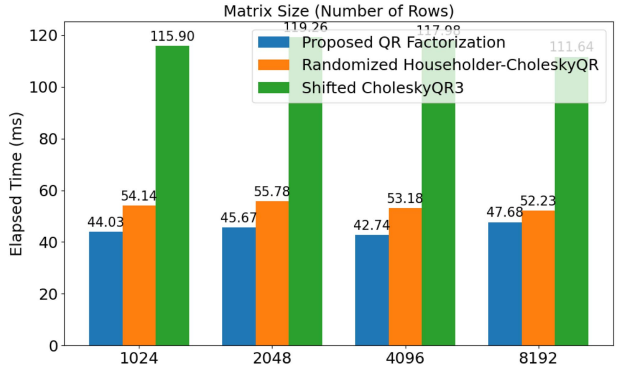


Fig. 3. Performance comparison of QR factorization using three different methods: Proposed QR Factorization, Randomized Householder-CholeskyQR, and Shifted CholeskyQR3. The x-axis represents the number of matrix rows, with all matrices having a fixed column size of 32.

V. Implementation and Optimization

Building on the performance analysis outlined above, we propose a hybrid approach that combines the recursive and TSQR variants based on the desired precision. In this section, we will detail our implementations and optimization techniques for FP64, FP32, and FP16 precision.

A. Panel Factorization Implementation

Building on the performance analysis of panel factorization discussed earlier, it is evident that regardless of the required precision, the tall and skinny QR factorization consistently outperforms conventional panel factorization. Consequently, this section will concentrate on the optimization of the tall and skinny QR factorization (TSQR) and the subsequent reconstruction of the WY representation after TSQR.

1) Tall and Skinny Panel QR Factorization: The TSQR algorithm discussed in the performance analysis appears to offer two distinct advantages:

1) Reduced Data Movement: Particularly beneficial when the panel QR does not fit in shared memory.
2) Shortened Critical Path: It effectively shortens the critical path, defined as the column-by-column sequence of processing.

To empirically demonstrate these advantages, we conducted some experiments, and Fig. 2 presents the performance of Householder TSQR against the cuSOLVER `SGEQRF` routine for matrix sizes $m \times 32$. In scenarios where m is small enough to fit the entire problem in the L1 cache of the GPU, TSQR exhibits marginal improvement. However, as the problem size surpasses the cache capacity, TSQR consistently outperforms `SGEQRF` with an increasing margin. This superiority is attributed to the combined effects of looser coupling and reduced data movement.

In Fig. 3, we also compare our proposed algorithm with two emerging methods: Randomized Householder-CholeskyQR [42] and Shifted CholeskyQR3 [29] on RTX 4090 GPU using FP64 precision. The matrix size is $m \times 32$, where m is as shown on the x-axis. For the Randomized Householder-CholeskyQR method, the size of the random matrix used is $k \times m$ with $k = 100$. Our proposed algorithm achieves an average improvement of 1.1x over the Randomized Householder-CholeskyQR method and 2.7x over the Shifted CholeskyQR3 method. In testing Shifted CholeskyQR3 with matrices having singular values that follow geometric, arithmetic, and Gaussian distributions, we observed that it becomes unsolvable when the matrix size is excessively large.

2) Engineering a Fixed-Size Block QR in a Threadblock: The TSQR algorithm effectively partitions an arbitrarily tall and skinny matrix into fixed-size tiles, facilitating the optimization of specific problems. In our approach, we present a highly hand-optimized 256∗32 Householder kernel designed for a single threadblock, and multiple threadblocks can be launched as needed. As these threadblocks operate independently, the optimization focus remains on individual threadblocks rather than their joint optimization.

The Householder QR factorization follows a column-by-column processing structure. In each column iteration, the algorithm computes the norm of the current column, scales it by the inverse norm, and subsequently utilizes the current column for inner product and scaling/subtraction operations with the remaining columns. The fundamental computations involve inner-products of columns and element-wise operations, such as scaling and subtraction. To effectively distribute the data and workload across the 32 warps in the threadblock, two approaches based on row or column distribution can be employed, as illustrated in Fig. 4.

In Fig. 4, the scheme on the left, used in [2] for Modified Gram-Schmidt, exhibits coordination challenges among all warps due to inner products being performed on columns. Additionally, when updating the remaining columns, all warps must sequentially process them. On the right, a more efficient mapping is presented, where all reductions are inter-warp, leveraging the `__shfl_down_sync` primitive for register exchange. This enables independent operation of warps when updating the remaining columns, reducing the number of reduction operations from 32 to 2. Consequently, the right scheme for Householder QR factorization is approximately 1.5x faster than the left scheme, despite involving more arithmetic operations in Householder QR factorization.

The performance of the GEMMs in TSQR can also benefit from Tensor Cores. By simply replacing batched GEMMs

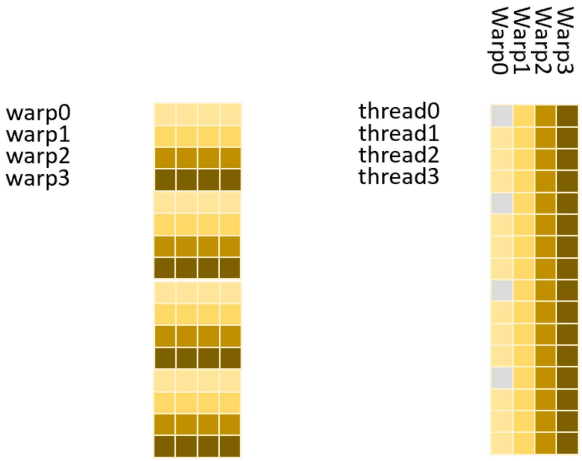


Fig. 4. Row-based data/work distribution (left) leads to inter-column dependencies, while column-based distribution (right) enables efficient shuffling in the QR factorization of a 256×32 tile.

with batched Tensor-Cores-based GEMMs, such as cuBLAS `cublasGemmStridedBatchedEx` routine, a satisfactory performance increase can be achieved.

3) Reconstruct Householder Vectors: Although TSQR can accelerate the panel compared to cuSOLVER `SGEQRF`, a challenge arises: unlike cuSOLVER `SGEQRF`, TSQR provides the explicit Q. In contrast, cuSOLVER `SGEQRF` yields Householder vectors. Moreover, using the explicit Q instead of Householder vectors in further trailing matrix updates may lead to unstable results. Therefore, developing an algorithm that can output Householder vectors from the explicit Q formed by TSQR is essential. One solution is to reconstruct Householder vectors from the explicit Q [43]. The idea is as follows: given an orthogonal matrix, Q can be represented as $Q = I - Y \times T \times Y^T$ (memory-efficient WY representation), and this equation can also be modified to $I - Q = Y \times T \times Y^T$. Here, Y is a lower triangular matrix, and T is an upper triangular matrix. Consequently, this can be regarded as an LU factorization with $I - Q = L \times U = (Y) \times (T \times Y^T)$. The paper [43] also reports that the LU factorization provides a unique L and U, and partial pivoting is unnecessary. In our algorithm, we factorize the upper $n \times n$ part of the matrix $I - Q$, and then we perform a triangular solve to obtain the complete L matrix. After obtaining the Householder vectors L from the LU factorization, we perform another triangular solve to construct W; refer to Algorithm 2.

Ballard et al. perform an LU factorization on $Q - S$ [43], where S is a diagonal sign matrix corresponding to the sign choices made inside the Householder QR algorithm. It's important to note that this step is essential to avoid rank deficiency in the LU factorization. By combining this algorithm with our TSQR implementation, we can achieve the panel factorization that generates W and Y.

Algorithm 2: Reconstruct WY Representation by Explicit Q Generated From TSQR.

```
function [W,Y] = ReconstructWY(Q)
  [m,n] = size(Q);
  I = eye(m,n);
  A = I-Q;
  [L1,U] = non_pivoting_LU(A(1:n,:));
  L2 = A(n+1:m,:)/U;
  Y=[L1;L2];
  W=A/Y';
end
```

B. Recursive-Like Triangular Matrix Multiplications

The trailing matrix update process in the iterative variant is the same as conventional GEMMs, where we use cuBLAS relative GEMM routines. However, for the recursive algorithm, the trailing matrix update, as well as forming the W matrix, involves the Householder matrix Y, which is a lower triangular matrix. Considering the triangular matrix multiplication (`TRMM`) as normal GEMMs is a waste of GPU resources since `TRMM` has fewer arithmetic operations than GEMMs. Unfortunately, there's no available `TRMM` routine on Tensor Cores. Thus, we introduce optimization techniques for `TRMM` in this section.

The `TRMM` operation also involves many GEMM operations during computation. The conventional `TRMM` is computed by the following formula [44]:

$$C_{ij} = \alpha \sum_{k=i+1}^{z} A_{ik} B_{kj} + A_{ii} B_{ij}$$

where i and j denote the block index in C. The $A_{ik}B_{kj}$ represent the GEMM operations, while $A_{ii}B_{ij}$ corresponds to the `TRMM` operation. However, when utilizing Tensor Cores, the conventional variant encounters the same problem: the inside GEMMs are not square and large enough if the blocksize is not carefully selected. In this case, we can also use the recursive variant for `TRMM`, and the formulation is expressed as follows:

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right] \tag{6}$$

$$\left[C_{11} \mid C_{12}\right] = \left[A_{11}\right] \left[B_{11} \mid B_{12}\right] \tag{7}$$

$$\begin{aligned} \left[C_{21} \mid C_{22}\right] &= \left[A_{21} \mid A_{22}\right] \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] \\ &= \left[A_{21}\right] \left[B_{11} \mid B_{12}\right] + \left[A_{22}\right] \left[B_{21} \mid B_{22}\right] \end{aligned} \tag{8}$$

Since A is a triangular matrix, A_{11} and A_{22} are also triangular matrices. As a result, the equations $[A_{11}][B_{11}|B_{12}]$ (7) and $[A_{22}][B_{21}|B_{22}]$ (8) can also be regarded as subproblems of `TRMM`, while $[A_{21}][B_{11}|B_{12}]$ (8) is a GEMM that can be executed on Tensor Cores. By using the recursive algorithm, the inside GEMMs can be 'compressed' to be more square and larger to be executed efficiently on Tensor Cores.

Unlike QR factorization, `TRMM` doesn't have strict data dependency, so most of the computations can be run in parallel. Thus, even though we're using the recursive formulation to illustrate our idea, the real implementation still uses iterations to bring more parallelism. Fig. 5 shows a visual representation of

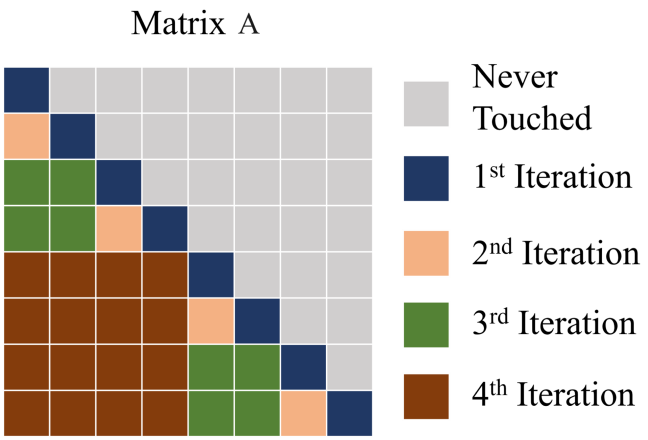


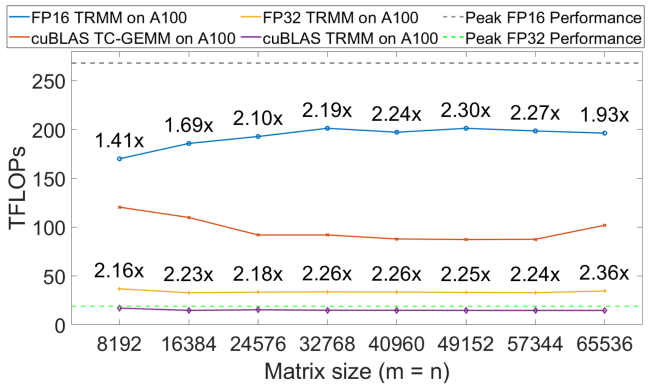
Fig. 5. The steps of computing recursive TRMM in parallel.

Algorithm 3: Recursive TRMM With Cutoff Size **nb**.

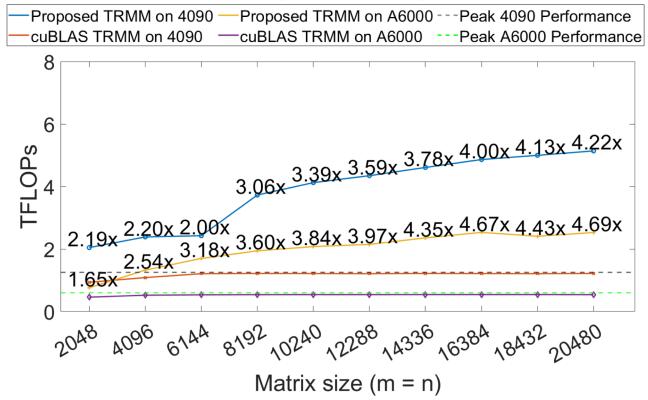
```
function [C] = tc_trmm(A)
  [n,k] = size(A);
  &percnt; BatchedGEMM((m,n,k),ptr(A),ptr(B),
     &percnt; offsetofA,offsetofB)
   &percnt; 1stiteration
  C = BatchedGEMM((nb,n,nb),A B,
                  nb*(1+lda),nb);
  i=1;
  while(m/nb/i/2>=1)
     &percnt; 2ndton − thiteration
    Ci =
    BatchedGEMM((i*nb,i*nb,k),A+i
    *nb,B,
                  2*i*nb*(1+lda), 2*i*nb);
    i=i*2;
  end
  C = Accumulate(C1,C2,...,Cn);
end
```

this process. The computations involving the diagonal blocks in matrix A are first executed, and the results are stored in matrix C for subsequent accumulations. Subsequently, the off-diagonal blocks of matrix A are processed sequentially. See Algorithm 3 for more implementation details. Note that this algorithm works for all three precisions; the only difference is the working precision in `BatchedGEMM` in Algorithm 3.

1) Performance Evaluation on TRMM: To illustrate the efficiency of the proposed `TRMM` algorithm, we conducted the same experiments on A100, RTX 4090, and A6000 GPUs to show the superior performance of our `TRMM` compared to cuBLAS. In the recursive variant, both $A_2 = (A_2 - Y_1 W_1^T A_2)$ and $W = [W_1 | - W_1 Y_1^T W_2 + W_2]$ (2) involve square `TRMM`. Therefore, we only provide the square `TRMM` performance here. Fig. 6(a) shows the FP16 and FP32 performance of the proposed `TRMM` using TC-GEMMs and TC-SGEMMs, respectively. For FP16 precision, compared to the `cublasGemmEx` routine,



(a) FP16 and FP32 TRMM performance comparison on A100 GPU



(b) FP64 TRMM performance comparison on RTX 4090 and A6000 GPU

Fig. 6. FP16/32 TRMM performance comparison on A100 GPU (upper figure); FP64 TRMM on RTX 4090 and A6000 GPU (lower figure)

our `TRMM` can achieve over 2x speedup when the matrix is large (ideally the speedup is 2x, as `cublasGemmEx` performs normal GEMMs, thereby doubling the arithmetic operations), and the extra speedup above 2x proves the better utilization of Tensor Cores. For FP32 `TRMM`, benefiting from TC-GEMMs, we are also able to achieve over 2x speedup compared to the `cublasStrmm` routine. Finally, on RTX 4090 and A6000 GPUs, because the FP64 precision computing capacities are limited on these GPUs, the TC-DGEMMs are over 4x faster than `cublasDgemm`. Thus, with the Tensor-Cores-based `TRMM` acceleration, the trailing matrix update process can be even faster in the recursive variant, and we will show the final QR factorization performance in the experiments section.

C. Other Optimization Techniques

There are also some other small but non-trivial optimization techniques that can also bring another 10%-20% speedup in terms of trailing matrix update, including data type conversion, kernel scheduling, and zero padding.

1) Fast Data Type Convert in FP16 Precision Implementation: While the `cublasGemmEx` routine supports FP32 inputs, the performance diminishes significantly when compared to utilizing FP16 inputs, notwithstanding the achievement of identical final accuracy. In order to optimize the performance of FP16 TC-GEMMs, we adopt a proactive approach by converting

the FP32 matrix to an FP16 matrix. The conversion process is primarily constrained by memory bandwidth rather than computational resources. Therefore, an efficient conversion process holds the potential to markedly reduce the overhead associated with TC-GEMM operations.

The conventional method for the data conversion process involves mapping one element conversion to one thread on the GPU. However, this method is hampered by the latency associated with device memory load, resulting in a degradation of throughput. To alleviate the latency stemming from device memory load, we employ optimization techniques, including software pipelining and the utilization of 128-bit device memory load/store instructions. This strategy entails partitioning the input matrix into submatrices, converting FP32 elements to FP16, and subsequently storing the FP16 submatrix back into device memory. This approach effectively utilizes up to 80% of the A100 GPU's device memory bandwidth, representing a noteworthy enhancement from the 55% utilization achieved with the naive method.

Based on multiple evaluations conducted on a half-precision matrix of size 65536×65536, the average total time taken for decomposition before applying fast data type convert was 3417.82ms, with the data type conversion taking 309.64ms. Following the implementation of this technique, both the total decomposition time and the duration for data type conversion were reduced to 3250.51ms and 142.33ms, respectively. Consequently, the data type conversion process constitutes approximately 10% of the time allocated for the trailing matrix update, while the enhanced data type conversion method contributes approximately a 5% acceleration to the overall trailing matrix update process.

2) CUDA Kernel Scheduling: In Algorithm 3, it is evident that batched GEMMs are utilized to augment parallelism in the recursive formulation. However, it is apparent that, during the iterations, the batched GEMMs are executed sequentially despite their potential for parallel execution. This lack of parallelism becomes particularly constraining for performance when dealing with small matrix sizes. To rectify this limitation, we have developed a custom kernel tailored for small matrix sizes, allowing us to exert control over the behavior of the CUDA kernel and maximize parallelism.

For example, in the first and second iterations of Algorithm 3, when dealing with relatively small matrices, we launch n/nb thread blocks to perform the batched `TRMM` and $n/nb/2$ thread blocks to execute the batched GEMMs in the second iteration. For subsequent computations, we leverage CUDA streams to synchronize their executions and activate Tensor Cores to the fullest extent possible. This approach ensures improved parallelism and better utilization of computational resources, especially when dealing with smaller matrix sizes.

According to our experiments, for a FP32 precision matrix size of 65536×65536, the average time without kernel scheduling optimization is 13153.24ms on A100 GPU, while the average decomposition time after applying CUDA kernel scheduling is 12218.22ms. Depending on the matrix size, this strategy can contribute an overall acceleration ratio of 5% to 10% to the algorithm.

3) Zero Padding: The most recent version of cuBLAS now accommodates arbitrary GEMM sizes on Tensor Cores; nevertheless, certain constraints continue to hinder complete utilization. In our experimentation on the A100 GPU, the performance of our `TRMM` operation on a matrix size of $30,000 \times 30,000$ attains 205 TFLOPs, whereas a matrix size of $30,001 \times 30,001$ yields only 18 TFLOPs. In response to this limitation, we have devised a strategy to augment overall performance by expanding the original matrix with zeros in the additional blocks. Specifically, for irregular matrices (with an odd number of rows and columns), we append a zero matrix to the bottom-right corner and fill the top-right and bottom-left areas with zeros, expanding the matrix so that both the row and column counts are multiples of 4. This approach enables more efficient utilization of Tensor Cores, mitigating the performance drop observed with non-optimized matrix sizes. For example, in the case of the previously mentioned 30,000 $\times$ 30,001 matrix, after applying zero padding to expand it to 30,000 $\times$ 30,004, testing showed that performance improved to approximately 180 TFLOPS. Depending on the shape of the matrix, this method can achieve nearly a 10x peak performance improvement. Considering the frequency of its application in our algorithm, we estimate its overall acceleration effect to be around 5%.

VI. Experimental Evaluation

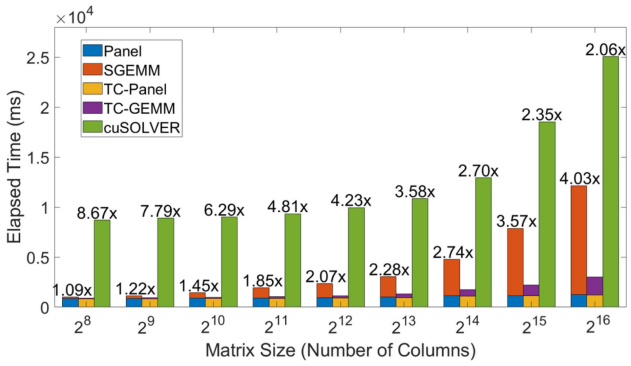
In this section, we perform a series of experiments to assess the performance of the proposed Householder QR factorization utilizing Tensor Cores across various precision modes, namely FP16, FP32, and FP64. The experiments are conducted on an A100-PCIe GPU with 80 GB of device memory for FP16 and FP32 precision, while an RTX 3090 GPU with 24 GB of device memory is employed for FP64 precision. The CUDA version utilized is 12.2, encompassing a C++ compiler along with cuBLAS and cuSOLVER libraries.

Furthermore, we conduct experiments to elucidate accuracy issues across all three precisions. Random matrices with diverse condition numbers and distributions are generated using MAGMA 2.7.2, providing a comprehensive evaluation of the proposed Householder QR factorization method under varying conditions.

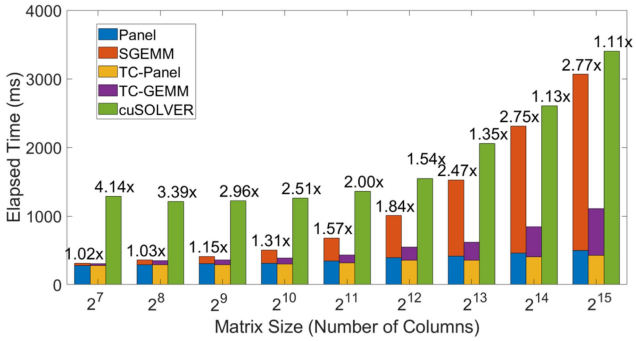
A. Performance Evaluation

Given that TC-DGEMMs require the utilization of INT8 Tensor Cores, while TC-SGEMMs and TC-GEMMs merely necessitate FP16 Tensor Cores, we structured our discussions based on Tensor Cores precision and architectures. Specifically, we selected two types of GPUs: A100-like GPUs (including GPUs such as H100 and V100), whose SGEMMs can be replaced by TC-SGEMMs, while their DGEMMs are faster than TC-DGEMMs; and RTX 3090-like GPUs (including GPUs such as RTX 4090 and A6000), which have a much weaker FP64 computing capacity than FP32 and FP16.

In Fig. 7, we depict the performance of our implemented FP16 and FP32 precision QR factorization using Tensor Core

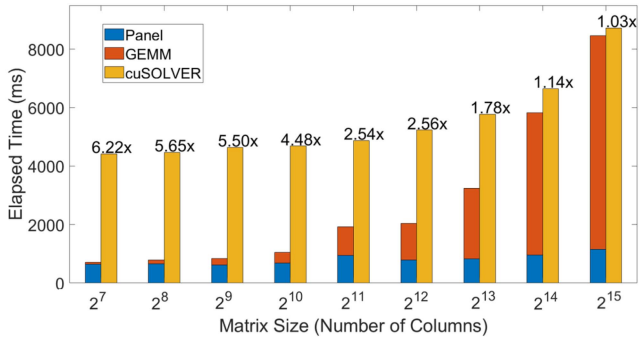


(a) FP16 and FP32 (Tensor Cores enabled) Householder QR factorization performance comparison on A100 GPU, the matrix size is $m \times n$, where $m * n = 2^{32}$

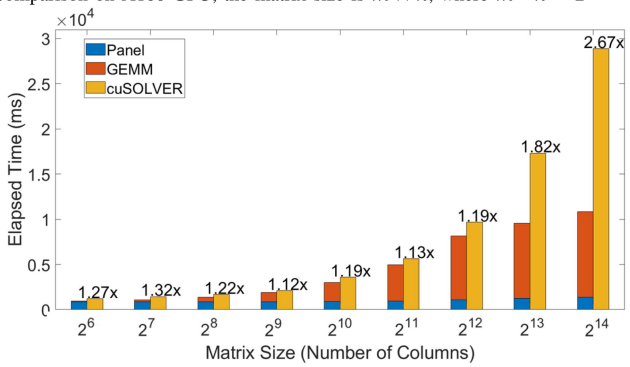


(b) FP16 and FP32 (Tensor Cores disabled) Householder QR factorization performance comparison on RTX 3090 GPU, the matrix size is $m \times n$, where $m * n = 2^{30}$

Fig. 7. Overall QR factorization performance comparison between the proposed FP16 QR, FP32 QR and cuSOLVER's FP32 QR. The numbers on cuSOLVER bars denote the speedup of the proposed FP32 QR and cuSOVLER's FP32 QR, and the numbers on FP32 QR bars denote the speedup of the proposed FP16 QR compared to the proposed FP32 QR



(a) FP64 (Tensor Cores disabled) Householder QR factorization performance comparison on A100 GPU, the matrix size is $m \times n$, where $m * n = 2^{31}$



(b) FP64 (Tensor Cores enabled) Householder QR factorization performance comparison on RTX 3090 GPU, the matrix size is $m \times n$, where $m * n = 2^{29}$

Fig. 8. Overall QR factorization performance comparison between the proposed FP64 QR, cuSOLVER FP32 QR on A100 and RTX 3090 GPU. The numbers on cuSOLVER bars denote the speedup of the proposed FP64 QR compared to cuSOVLER's FP64 QR

alongside cuSOLVER routines on A100 GPU [4] and RTX 3090 GPU,[5] as well as the speedup (the numbers on the 3rd bar denote the speedup of our FP32 QR compared to cuSOLVER, the numbers on the 1st bar denote the speedup of our FP16 QR compared to our FP32 QR) of our approach compared to cuSOLVER relative routines. The matrix sizes, from left to right, represent the number of columns (n), while the number of rows (m) decreases from 2^{24} to 2^{16}. In other words, the matrix shapes change from tall and skinny to square.

On one hand, when the matrix is square or nearly square, the acceleration is mainly from GEMMs using Tensor Cores; therefore, the FP16 QR is 4.03x faster than the proposed FP32 QR. Also, compared to cuSOLVER `SGEQRF`, our FP32 can still have a 2.06x speedup benefit from TC-SGEMMs in the recursive variant. On the other hand, when the matrix is tall and skinny, the TSQR contributes more to the performance, while the Tensor Cores cannot bring large acceleration. That is the reason why the FP16 QR is only 1.09x faster than FP32 QR, but compared to cuSOLVER `SGEQRF`, the proposed FP32 QR exhibits significant acceleration at a factor of 8.67x.

The impact of the TSQR is more pronounced on the RTX 3090 GPU (Fig. 7(b)), as only TSQR involves FP32 precision. When the matrix is tall and skinny, the proposed FP32 QR is 4.14x faster than cuSOLVER. On the contrary, as Tensor Cores are not utilized, the acceleration is only 1.11x, which is from TSQR.

The FP64 precision QR factorization performance exhibits slight differences compared to FP32 and FP16 precision on both A100 and RTX 3090 GPUs (Fig. 8 [6]). On the A100 GPU, the DGEMMs utilizing INT8 Tensor Cores do not outperform cuBLAS DGEMMs. Consequently, the acceleration is solely derived from TSQR, with tall and skinny matrices achieving up to a 6.22x speedup compared to the cuSOLVER `DGEQRF` routine, as illustrated in Fig. 8(a). On the RTX 3090 GPU, TC-DGEMMs can be as much as 7x faster than cuBLAS DGEMMs. Even for square matrices, the proposed FP64 QR factorization can

[4]From left to right bars, the TFLOPs of the proposed FP16 and FP32 QR factorization are [2.4, 4.7, 8.9, 16.7, 30.4, 50.6, 70.6, 107.3, 109.8] for FP16 precision (peak 312.0), and [2.2, 3.9, 6.1, 9.1, 14.3, 21.0, 26.6, 30.4, 28.5] for FP32 precision (peak 19.5)

[5]From left to right bars, the TFLOPs of the proposed FP16 and FP32 QR factorization are [0.4, 0.8, 1.6, 3.1, 5.7, 10.2, 16.0, 26.2, 35.0, 37.4] for FP16 precision (peak 71.0), and [0.5, 0.9, 1.5, 2.7, 4.4, 6.5, 8.7, 11.3, 14.0, 15.3] for FP32 precision (peak 35.5)

[6]From left to right bars, the TFLOPs of the proposed FP64 QR factorization are [0.1, 0.2, 0.5, 0.9, 1.8, 3.4, 6.0, 10.1, 13.4] on A100 GPU (peak 19.5), and [0.03, 0.06, 0.10, 0.16, 0.26, 0.31, 0.39, 0.45, 0.47. 0.51] RTX 3090 GPU (peak 0.56)

achieve a 2.67x speedup, as shown in Fig. 8(b). However, it's crucial to address the performance disparity regarding tall and skinny matrices on the RTX 3090. Compared to the A100 GPU, the tall and skinny speedup is much smaller (1.27x vs. 6.22x). This discrepancy arises due to significant variations in peak FP64 performance between these two GPUs (19.5 TFLOPs vs. 0.55 TFLOPs). Consequently, on the RTX 3090 GPU, communication overhead is more manageable to some extent. Additionally, it's noteworthy that cuSOLVER DGEQRF is not well-optimized on the A100 GPU, utilizing only 6‰ of FP64 peak performance, while on the RTX 3090 GPU, the utilization percentage is 6%.

To sum up, our approaches showcase versatility, delivering significant acceleration across different types of GPUs and computation precisions. Both TSQR and Tensor Cores contribute to enhancing the QR factorization performance. The distinction lies in the fact that TSQR plays a more substantial role when dealing with tall and skinny matrices, whereas Tensor Cores exhibit greater significance in the context of square matrices. This adaptability underscores the effectiveness of our proposed methodologies in optimizing QR factorization across diverse scenarios and hardware configurations.

B. Error Analysis and Evaluation

There are two types of errors in QR factorization: backward error, defined as $\frac{||A-\hat{Q}\hat{R}||}{||A||}$; and orthogonality error, defined as $\frac{||I-\hat{Q}^T\hat{Q}||}{N}$. Extensive studies have been conducted to address QR factorization errors [45], and the error bounds of Gram-Schmidt and Householder QR factorization are presented in Table II. In the context of the proposed QR factorization in this paper, two main differences compared to conventional Householder QR factorization are highlighted: 1) the replacement of panel QR factorization with TSQR; and 2) the utilization of Tensor-Cores-based GEMMs.

The reconstruction of WY representation in TSQR has been proven to have the same error bound as conventional panel QR factorization [43]. Moreover, studies on the errors introduced by Tensor Cores [46] demonstrate that they have a reliable error bound compared to pure FP16 GEMMs, as Tensor Cores use FP32 precision for accumulation. Additionally, SGEMMs and DGEMMs using Tensor Cores also exhibit the same error bounds as default cuBLAS S/DGEMMs [20], [21].

Therefore, the backward error in the proposed Householder QR factorization will be bounded by:

$$E_b = \frac{||A - \hat{Q}\hat{R}||_F}{||A||_F} \leq c\epsilon_p$$

where c is a constant number, and ϵ_p is the unit round-off error of different precisions. The orthogonality error is bounded by:

$$E_o = \frac{||I - \hat{Q}^T\hat{Q}||_F}{N} \leq c\epsilon_p$$

Empirically speaking, the errors in FP16, FP32, and FP64 precision are close to $1e-4$, $1e-6$, and $1e-16$, respectively.

Table IV displays the experimental backward and orthogonality errors for various types of matrices, each with a size of 4096×4096. These matrices encompass uniform distribution, normal distribution, SVD arithmetic distribution, and SVD geometric distribution (condition number $1e4$). It is evident from the table that both the backward errors and orthogonality errors fall within the bounds of unit round-off error.

TABLE IV
THE ACCURACY RESULTS OF THE PROPOSED QR FACTORIZATION ON DIFFERENT PRECISIONS AND MATRIX TYPES ON A100 GPU (FP16 AND FP32) AND RTX 3090 GPU (FP64) USING TENSOR CORES

Error \ Matrix	Uniform	Normal	Arith	Geo
FP64 Backward	8.9E-16	1.3E-15	1.8E-15	2.5E-15
FP64 Orthogonality	9.0E-17	1.3E-16	1.7E-16	2.5E-16
FP32 Backward	7.6E-07	8.5E-07	1.3E-06	1.9E-06
FP32 Orthogonality	3.1E-07	3.8E-07	4.7E-07	6.3E-07
FP16 Backward	5.1E-04	4.3E-04	5.4E-04	6.4E-04
FP16 Orthogonality	8.7E-05	9.2E-05	9.1E-05	9.3E-05

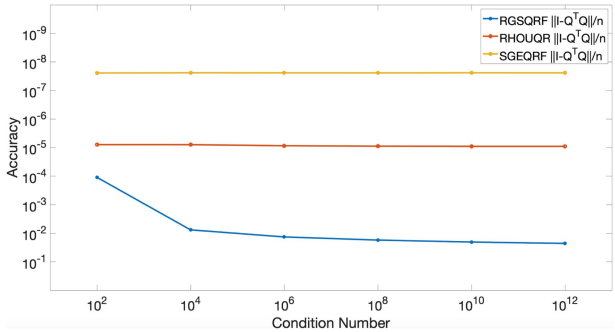


Fig. 9. QR factorization orthogonality accuracy: FP16 recursive Gram-Schmidt QR (RGSQRF) vs. FP16 recursive Householder QR (RHOUQR) vs. cuSOVLER SGEQRF, matrix size 8192∗4096, SVD geometric distribution.

In comparison to [2], which implements Gram-Schmidt QR factorization, the Householder QR factorization demonstrates significantly superior orthogonality. Fig. 9 illustrates that cuSOLVER SGEQRF and the proposed FP16 recursive Householder QR factorization exhibit full orthogonality up to working precision. In contrast, the FP16 recursive Gram-Schmidt QR factorization (RGSQRF) experiences a linear degradation in orthogonality with the increase of $\kappa(A)$. Although there exists a remedy for RGSQRF to restore its orthogonality error up to working precision, known as re-orthogonalization, our additional experiments unfortunately reveal that re-orthogonalization is effective only for simpler cases, such as matrices with SVD arithmetic distribution, and not for those with geometric distribution.

More specifically, the improved orthogonality results in a more effective preconditioner for solving linear least square problems (LLS solver) (Section III-B in our prior paper [2] elaborates on the direct method and the iterative method). Fig. 10 presents a performance comparison between the direct method using cuSOLVER SGEQRF and the iterative method using RGSQRF and RHOUQR as preconditioners. As per our earlier investigation (Section IV-B.1 in [2]), for simpler cases characterized by small condition numbers and favorable singular value distributions, employing Gram-Schmidt QR factorization as a preconditioner is sufficient to achieve satisfactory LLS solver performance compared to the direct method using cuSOLVER.

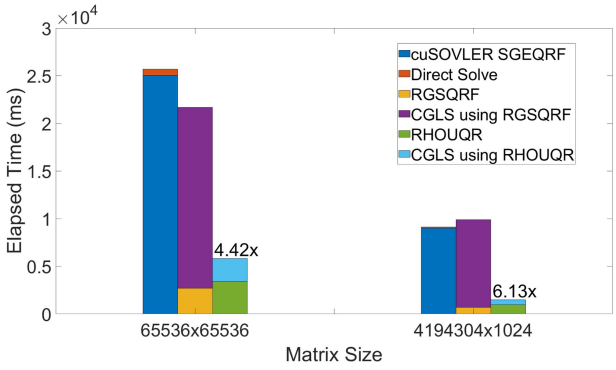


Fig. 10. The performance comparison on solving single precision linear least square problem using direct method (cuSOVLER SGEQRF) and iterative methods using RGSQRF and RHOUQR as preconditioner. The matrix sizes are 65536×65536 and 4194304×1024 and the condition number is 1e6 with SVD geometric distribution on A100 GPU.

However, in the face of more challenging matrices, such as those with SVD geometric distribution and a condition number of $1e6$, the CGLS with RGSQRF preconditioner becomes considerably more computationally expensive than in simpler cases, and at times, it is slower than the direct method. In contrast, the proposed FP16 precision Householder QR factorization offers a significantly improved preconditioner, resulting in an overall speedup of 4.42x and 6.13x compared to the direct LLS solver for matrix sizes 65536×65536 and 4194304×1024, respectively.

VII. Conclusion and Future Work

In this paper, we delve into the efficient implementations of QR factorization with a focus on tall and skinny matrices, incorporating Tensor Cores acceleration. The Householder QR factorization is partitioned into two variants: one involves QR factorization without Tensor Cores, utilizing FP64 precision on A100 GPU and FP32 precision on RTX 3090 GPU; the other employs Tensor Cores, encompassing FP16 and FP32 precision on A100 GPU and FP16 and FP64 on RTX 3090 GPU. For the non-Tensor-Cores QR factorization, only TSQR is employed, as TC-S/DGEMMs do not yield any speedup in the trailing matrix update process. In the Tensor-Cores-based QR factorization, TSQR and the recursive variant are utilized, as the recursive approach can facilitate larger and square GEMMs to capitalize on the benefits of Tensor Cores.

In comparison to our prior work, this paper extends the modified Gram-Schmidt QR factorization to Householder QR factorization and leverages customized Tensor-Cores-based `TRMM` to exploit the triangular property of Householder matrices. Additionally, we expand the half precision to include single and double precision by harnessing Tensor Cores to accelerate SGEMMs and DGEMMs.

Our future endeavors will focus on scaling the problem to factorize much larger matrices. Conventional scalar QR factorization routines, such as ScaLAPACK [47], MAGMA, and cuSOVLERMG,[7] are not compatible with Tensor Cores. With Tensor Cores in play, the bottleneck may shift from compute-bound to memory-bound [48]. This implies that the overall workflow of QR factorization needs to be redesigned to mitigate the time cost associated with data movement during computations.

[7]https://docs.nvidia.com/cuda/cusolver/index.html#using-the-cusolvermg-api

References

[1] 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 *Proc. IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 603–613.

[2] S. Zhang, E. Baharlouei, and P. Wu, "High accuracy matrix computations on neural engines: A study of QR factorization and its applications," in *Proc. 29th Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2020, pp. 17–28.

[3] S. Zhang, R. Shah, H. Ootomo, R. Yokota, and P. Wu, "Fast symmetric eigenvalue decomposition via WY representation on tensor core," in *Proc. 28th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, 2023, pp. 301–312.

[4] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding QR decomposition for GPUs," in *Proc. 2011 IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 48–58.

[5] T. NVIDIA, "Nvidia tesla v100 GPU architecture," NVIDIA, Tech. Rep. 1, 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[6] J. Burgess, "Rtx on—the nvidia turing GPU," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, Mar./Apr. 2020.

[7] NVIDIA, "Nvidia h100 tensor core GPU architecture," NVIDIA, Tech. Rep. 1, 2023. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core

[8] J. Choquette, "NVIDIA Hopper h100 GPU: Scaling Perform," *IEEE Micro*, 2023.

[9] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *Proc. 2020 IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 634–643.

[10] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," 2018, *arXiv: 1804.06826*.

[11] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021.

[12] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 359–371.

[13] Y. Wang, B. Feng, and Y. Ding, "QGTC: Accelerating quantized graph neural networks via GPU tensor core," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 107–119.

[14] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "APNN-TC: Accelerating arbitrary precision neural networks on ampere GPU tensor cores," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–13.

[15] F. Lopez and T. Mary, "Mixed precision LU factorization on GPU tensor cores: Reducing data movement and memory footprint," in *Proc. Int. J. High Perform. Comput. Appl.*, 2020, Art. no. 10943420221136848.

[16] A. Sorna, X. Cheng, E. D'azevedo, K. Won, and S. Tomov, "Optimizing the fast fourier transform using mixed precision on tensor core hardware," in *Proc. 2018 IEEE 25th Int. Conf. High Perform. Comput. Workshops*, 2018, pp. 3–7.

[17] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-M. Hwu, "Accelerating reduction and scan using tensor core units," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 46–57.

[18] X. Liu et al., "Toward accelerated stencil computation by adapting tensor core unit on GPU," in *Proc. 36th ACM Int. Conf. Supercomputing*, 2022, pp. 1–12.

[19] S. Zhang, R. Shah, and P. Wu, "Tensorsvm: Accelerating kernel machines with tensor engine," in *Proc. 34th ACM Int. Conf. Supercomputing*, 2020, pp. 1–11.

[20] H. Ootomo and R. Yokota, "Recovering single precision accuracy from tensor cores while surpassing the FP32 theoretical peak performance," *Int. J. High Perform. Comput. Appl.*, vol. 36, no. 4, pp. 475–491, 2022.

[21] H. Ootomo, K. Ozaki, and R. Yokota, "DGEMM on integer matrix multiplication unit," 2023, *arXiv:2306.11975*.

[22] C. Bischof and C. Van Loan, "The WY representation for products of householder matrices," *SIAM J. Sci. Statist. Comput.*, vol. 8, no. 1, pp. s2–s13, 1987.
[23] S. Tomov, R. Nath, P. Du, and J. Dongarra, "Magma users' guide," ICL UTK, 2011. [Online]. Available: https://icl.utk.edu/projectsfiles/magma/docs/magma-v02.pdf
[24] Z. Drmač, "Algorithm 977: A QR–preconditioned QR SVD method for computing the SVD with high accuracy," *ACM Trans. Math. Softw.*, vol. 44, no. 1, pp. 1–30, 2017.
[25] P. Luszczek, H. Ltaief, and J. Dongarra, "Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures," in *Proc. 2011 IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 944–955.
[26] M. Gates, S. Tomov, and J. Dongarra, "Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs," *Parallel Comput.*, vol. 74, pp. 3–18, 2018.
[27] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR factorization algorithms," in *Proc. 2011 Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.
[28] I. Yamazaki, S. Tomov, and J. Dongarra, "Mixed-precision cholesky QR factorization and its case studies on multicore CPU with multiple GPUs," *SIAM J. Sci. Comput.*, vol. 37, no. 3, pp. C307–C330, 2015.
[29] T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto, and Y. Yanagisawa, "Shifted cholesky QR for computing the QR factorization of ill-conditioned matrices," *SIAM J. Sci. Comput.*, vol. 42, no. 1, pp. A477–A503, 2020.
[30] P.-G. Martinsson, G. Quintana-Ortí, N. Heavner, and R. van de Geijn, "Householder QR factorization with randomization for column pivoting (HQRRP)," *SIAM J. Sci. Comput.*, vol. 39, no. 2, pp. C96–C115, 2017.
[31] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang, "Communication avoiding rank revealing QR factorization with column pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 36, no. 1, pp. 55–89, 2015.
[32] E. Anderson et al., *LAPACK Users' Guide*, vol. 9. Philadelphia, PA, USA: SIAM, 1999.
[33] D. B. Gajić, R. S. Stanković, and M. Radmanović, "A performance analysis of computing the LU and the QR matrix decompositions on the CPU and the GPU," *Int. J. Reasoning-Based Intell. Syst.*, vol. 9, no. 2, pp. 114–121, 2017.
[34] H. Ootomo and R. Yokota, "TSQR on tensor cores," in *Proc. 29th Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, Research Poster, 2019, doi: 10.1145/1122445.1122456.
[35] D. Vanderstraeten, "A stable and efficient parallel block gram-schmidt algorithm," in *Proc. Euro-Par'99 Parallel Process.: 5th Int. Euro-Par Conf.*Toulouse, France, Springer, Aug. 31–Sep. 03, 1999, pp. 1128–1135.
[36] E. Carson et al., "An overview of block gram-schmidt methods and their stability properties," 2020, *arXiv: 2010.12058*.
[37] L. Giraud, J. Langou, M. Rozložník, and J. V. D. Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numerische Mathematik*, vol. 101, no. 1, pp. 87–100, 2005.
[38] A. Kiełbasiński, "Analiza numeryczna algorytmu ortogonalizacji Grama-Schmidta," *Mathematica Applicanda*, vol. 2, no. 2, pp. 15–35, 1974.
[39] Å. Björck, "Solving linear least squares problems by Gram-Schmidt orthogonalization," *BIT Numer. Math.*, vol. 7, no. 1, pp. 1–21, 1967.
[40] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *Proc. 2018 IEEE Int. parallel Distrib. Process. Symp. Workshops*, 2018, pp. 522–531.
[41] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[42] A. J. Higgins, D. B. Szyld, E. G. Boman, and I. Yamazaki, "Analysis of randomized householder-cholesky QR factorization with multisketching," 2023, *arXiv:2309.05868*.
[43] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik, "Reconstructing householder vectors from tall-skinny QR," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1159–1170.
[44] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing," in *Proc. 2016 Int. Conf. Supercomputing*, 2016, pp. 1–11.
[45] M. P. Connolly and N. J. Higham, "Probabilistic rounding error analysis of householder QR factorization," *SIAM J. Matrix Anal. Appl.*, vol. 44, no. 3, pp. 1146–1163, 2023.
[46] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores," *SIAM J. Sci. Comput.*, vol. 42, no. 3, pp. C124–C141, 2020.
[47] L. S. Blackford et al., *ScaLAPACK Users' Guide*, vol. 4. Philadelphia, PA, USA: SIAM, 1997.
[48] S. Zhang and P. Wu, "Recursion brings speedup to out-of-core tensorcore-based linear algebra algorithms: A case study of classic gram-schmidt QR factorization," in *Proc. 50th Int. Conf. Parallel Process.*, 2021, pp. 1–11.

Yuhan Leng is currently working toward the master's degree in computer science and technology with the University of Electronic Science and Technology of China. Her research interests primarily include high performance computing and numerical linear algebra.

Gaoyuan Zou received the bachelor's degree from Southwest Jiaotong University. He is currently working toward the master's degree with the School of Computer Science and Engineering, University of Electronic Science and Technology of China. His interests lie in high performance computing and numerical linear algebra.

Hansheng Wang received the engineering bachelor's degree from North China Electric Power University, in 2011. He is currently working toward the PhD degree with the School of Computer Science and Engineering, University of Electronic Science and Technology of China. His research interests include high-performance computing, artificial intelligence, and the acceleration and practical application of large models.

Panruo Wu received the PhD degree from the University of California Riverside in 2016. He is an associate professor of computer science with the University of Houston. Prior to joining UH, he was a postdoc research with Innovative Computing Laboratory at University of Tennessee Knoxville. His research interests include matrix/tensor computations and numerical optimizations with large scale and on accelerators and he received NSF CAREER award, in 2021.

Shaoshuai Zhang received the PhD degree from the University of Houston in 2021. He is an assistant professor of School of Computer Science and Engineering, University of Electronic Science and Technology of China. Prior to joining UESTC, he was a postdoc research with Yokota Lab with the Tokyo Institute of Technology. His research interests include high performance computing, numerical linear algebra and numerical methods.