High Accuracy Matrix Computations on Neural Engines: A Study of QR Factorization and its Applications

Shaoshuai Zhang, Elaheh Baharlouei, Panruo Wu
Department of Computer Science
University of Houston
{szhang36,ebaharlouei,pwu7}@uh.edu

Abstract

Fueled by the surge of ever expanding successful applications of deep neural networks and the great computational power demanded, modern computer processors and accelerators are beginning to offer half precision floating point arithmetic support, and special units (neural engines) such as NVIDIA TensorCore on GPU and Google Tensor Processing Unit (TPU) to accelerate the training and prediction of deep neural networks. It remains unclear how neural engines can be profitably used in applications other than neural networks. In this paper we present an endeavor of accelerating and stabilizing a fundamental matrix factorization on neural engines—the QR factorization—which may open doors to much wider relevance to scientific, engineering, and data science. We show that traditional Householder QR algorithms and implementations do not have the necessary data locality, parallelism, accuracy, and robustness on neural engines which are characterized by extreme speed and low precision/range.

We demonstrate that neural engines can be effectively used to accelerate matrix computations (QR 3.0x-14.6x speedup compared to cuSOLVER, reaching up to 36.6TFLOPS); however different algorithms (recursive Gram-Schmidt) are needed to expose more locality and parallelism, even at the cost of increased computations. Moreover, scaling, iterative refinement, and other safeguarding procedures are also needed to regain accuracy and avoid overflowing. Our experience seems to suggest that presently with neural engines the matrix factorizations (QR, LU, Cholesky) are best to be co-designed with its applications (linear solver, least square, orthogonalization, SVD, etc) to achieve high performance and adequate accuracy and reliability.

CCS Concepts

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; **Computations on matrices**; • **Theory of computation** → **Parallel algorithms**; *Preconditioning*; • **Computer systems organization** → *Neural networks.*

ACM Reference Format:
Shaoshuai Zhang, Elaheh Baharlouei, Panruo Wu. 2020. High Accuracy Matrix Computations on Neural Engines: A Study of QR Factorization and its Applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392685

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392685

1 Introduction

Driven by the need to train large scale deep neural networks, there's been a tidal wave of the specialized low precision matrix matrix multiplication units. Among them are TensorCore from NVIDIA on its Volta and Turing architecture, Google's Tensor Processing Unit (TPU)[1], and Intel's latest FPGA, CPU, and Nervana Neural Processors. These neural engines are usually characterized by the support of lower precision arithmetic (such as 16-bit floating point format), and extremely efficient matrix-matrix multiplication. For example, NVIDIA V100 boasts up to 120 "deep learning" TeraFLOPS (120×10^{12} floating point operation per second) [32], which is half precision matrix multiplication accumulated in single precision. Google's TPU v3 claims 420 TeraFLOPS, also in doing half precision matrix-matrix multiplication. In contrast, V100 single precision peak performance is 14 TeraFLOPS, and double precision is 7TeraFLOPS. Having these neural engines greatly speeds up applications that primarily spend time in low precision matrix-matrix multiplication, and also results in much higher energy efficiency.

However, outside the neural networks, the effective use of such neural engines is only emerging. Two challenges must be addressed in using neural engines: 1) how to expose enough locality and parallelism to enable neural engines to run at high speed? 2) how to mitigate the loss of accuracy of using the limited half precision format? In this paper, we present an effective use of NVIDIA TensorCore units to QR factorize matrix and its applications in solving linear least square problems, orthogonalization, and low rank approximation. Least square problem and its many variants are prevalent in science, engineering, and statistical machine learning; for instance, non-linear least square problems are probably the largest source of all non-linear optimization problems. As such, QR factorization and its applications form a core component of any linear algebra packages such as LAPACK [1] which have been downloaded millions of times, and supported by all major processor vendors.

Thus, we set to answer the following question: is it profitable to use neural engines to accelerate common linear algebra operations reliably and accurately? We use QR factorization to demonstrate that the answer is yes, but new algorithms are needed to satisfy the data locality and parallelism that neural engines need to run at full speed and to compensate for the loss of accuracy and stability.

We consider the contributions of this paper to be:

[1] https://cloud.google.com/tpu/

- We propose novel recursive Gram-Schmidt QR that exhibits better data locality at the cost of doing more computations. The net result is vastly faster QR (3.0x to 14.6x, reaching 36.6TFLOPS) compared to single precision cuSOLVER QR (SGEQRF).
- We devise cheap iterative refinement, re-orthogonalization, and automatic column scaling to regain accuracy and avoid overflow.
- We conduct a comprehensive empirical study of the accuracy and performance of QR factorization, least square solver, and orthogonalization for a variety of matrices, with different sizes, aspect ratio, and spectrum distribution.

The paper is organized as follows. Section 2 introduces numerical, algorithmic, and the architectural background to understand this paper. Section 3 introduces the main methods, analysis, and rationale behind our algorithmic design and implementation. Section 4 is a comprehensive empirical study on the accuracy and performance of the proposed methods. Section 5 discusses related work and the context around this paper, and section 6 wraps up it with a conclusion.

2 Background

In this section, we review some background that are most relevant to understand this paper.

2.1 Half Precision Arithmetic and TensorCore GPU

NVIDIA introduced a specialized unit called TensorCore from their Volta architecture, which boasts up to 120 TFLOPS (120×10^{12} floating point operations per second) for half precision (FP16) matrix-matrix multiplication. Compared to single precision SGEMM (Single precision GEneral Matrix-Matrix multiplication) and double precision DGEMM, TensorCore is 7x and 14x faster respectively, which is a considerable upgrade in the performance at the cost of significantly lower precision and consequent loss of accuracy and numerical stability.

TensorCore only supports matrix-matrix multiplication (GEMM[2]). The easiest to use API is from cuBLAS, and it has many variations. A more flexible and also highly efficient way to program TensorCore is through the CUTLASS template library[3] from NVIDIA, or directly call the WMMA intrinsic. For this paper, we use TensorCore through cuBLAS library.

The Google Tensor Processing Unit (TPU) also depends extensively on 16 bits floating point matrix-matrix multiplication to achieve its claimed 420 TFLOPS in its latest TPU v3 offering. However, the 16 bits floating point format TPU uses is slightly different from the NVIDIA TensorCore; TPU uses the bfloat16 format, which has 3 fewer bits for mantissa and use 3 more bits for exponents so it can represent a wider range of numbers at a lower resolution. Intel also planned to introduce bfloat16 processing (together with FP32 accumulation) in their future processors (Cooper Lake Xeon) so we will see more variety of half precision support in mainstream processors, which makes it even more useful to extend the use pattern of low precision computing beyond deep neural networks.

[2]LAPACK subroutine naming convention: `SGEMM` means single precision general matrix multiplication, and `DGEMM` means double precision one

[3]https://github.com/NVIDIA/cutlass

Let us take a look at the different floating point format and see what gives and what takes in terms of accuracy (resolution in representing real numbers), and range (smallest and largest representable real number) The IEEE single precision floating point format is accurate and widely ranged, for it has 32 bits to spare. There are currently two widely implemented 16 bits floating point formats. Among them, IEEE FP16 has a significantly constrained range, but its resolution (the unit roundoff error—the distance to the next representable number from 1) is about 10 times better than bfloat16. bfloat16, on the other hand, has the same range as single precision, but its resolution is very limited (there is no bfloat16 number between 1 and 1.0078). So bfloat16 is more robust (less prone to overflow and underflow) but less stable/precise (large roundoff error). In this paper, we use FP16 format supported by NVIDIA TensorCore.

2.2 Linear Least Square (LLS) Problems

LLS problems are prevalent in natural science, engineering, and data science. To give a specific example (gradiometry), consider the large scale least square problems solved today concerning the determination of the Earth's gravity field from highly accurate satellite measurements [16]. Another example is the least square problems arising from many fields (data fitting, statistical machine learning, geodesy, computer vision, robotics (bundle adjustment), etc). Non-linear least square problems can often be solved as a series of linear least square problems.

The (over-determined) linear least square problem is stated as an minimization problem:

$$\min_x ||Ax - b||^2 \tag{1}$$

where $A \in \mathbb{R}^{m \times n} (m > n)$ has full column rank, and $b \in \mathbb{R}^n$. Geometrically, this minimization is to find the "projection" of point b onto the range (column space) of matrix A. Analytically the LLS problem has closed form solution:

$$x^* = (A^T A)^{-1} A^T b \tag{2}$$

Computationally, the analytical solution can be obtained by solving the square linear equation (called the normal equation): $A^T Ax = A^T b$. Typically a Cholesky factorization of $A^T A = LL^T$ can lead to a solution, via backward and forward substitution; this method is called normal equations. However directly forming $A^T A$ is unstable for all but the most well-conditioned systems; in practice we would avoid forming $A^T A$ directly.

A much more reliable and accurate method is based on QR factorization. For a tall and skinny matrix A it takes roughly 2x flops than the normal equations method, but it handles a much wider range of matrices. The basic idea is as follows. First we factorize the rectangular matrix $A \in \mathbb{R}^{m \times n}, m > n$ into the product of an orthogonal matrix $Q \in \mathbb{R}^{m \times n}$, and a square upper triangular matrix $R \in \mathbb{R}^{n \times n}$: $A = QR$. Then the solution to (1) is given by the following elementary matrix-vector operation:

$$x^* = R \backslash (Q^T b) \tag{3}$$

The aforementioned direct solvers are robust but could be slow for large scale problems. Iterative methods are more attractive for large scale and especially sparse problems, where the only operation involving matrix A is the matrix-vector multiplication Av and $A^T v$.

Terminology	Definition
LLS	Linear Least Square
TFLOPS	Tera floating point operations per second.
cond(A)/$\kappa(A)$	Condition number of a matrix A
cuSOLVER	A CUDA library from NVIDIA that provides LAPACK-like direct solvers
SGEMM	Single precision General matrix multiplication
TC-GEMM	TensorCore General matrix multiplication
GEMV	General matrix vector multiplication
[S/D]GEQRF	Single/Double precision General matrix Householder QR Factorization
[S/D]ORMQR	Orthogonal Multiply by Q from QR factorization: Perform QC or Q^TC with the implicit Q formed by Householder QR.
[S/D]TRSM	Single/Double precision; solve a triangular system.
CGLS	Conjugate Gradient method for Least Square
CAQR	Communication Avoiding QR factorization
RGSQRF[4]	Recursive Gram-Schmidt QR Fatorization
[S/D]CuSOLVE	Single/Double LLS direct solver ([S/D]GEQRF+[S/D]ORMQR+[S/D]TRSM)
[S/D]SOLVE	Single/Double solve $QRx = b$ (GEMV+[S/D]TRSM)

Table 1: Definitions of Terminology and Abbreviations, the first block is the general abbreviations, the second block is the commonly used routines, the third block is the methods used in our work

In this paper, we are going to combine the direct solver based on QR factorization (preconditioning), with an iterative method as safeguards to refine accuracy (iterative refinement). The hope is that we can get the best of both worlds—the opportunity to use TensorCore and predictability/stability of direct methods, and flexibility to take an inaccurate solution/factorization and turn it into an increasingly accurate solution through iterations.

2.3 Terminology or Abbreviations

To make this paper more readable, the definitions of terminology are given in Table 1.

3 Methods

In this section, we set out to design and engineer a TensorCore accelerated QR factorization subroutine, and devise iterative refinement, re-orthogonalization, and scaling to regain accuracy and reliability lost due to half precision TensorCore under the context of LLS, orthogonalization, and optimal low rank approximation. We also provide error analysis to guide the use of TensorCore accelerated QR and the various accuracy improvement techniques.

3.1 Accelerating QR factorization on TensorCore

As briefly introduced in section 2.2, QR factorization is one of the most fundamental matrix factorizations in numerical linear algebra. It seeks to factorize a general matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R. QR is almost always an important building block of any numerical linear algebra packages such as LAPACK [1], ScaLAPACK[7]. On GPU, NVIDIA provides well-optimized cuBLAS for basic matrix operations such as multiplication, and cuSOLVER for high-level matrix factorizations, such as LU/QR and eigendecompositions. A more comprehensive package is the MAGMA [14], which uses a hybrid CPU/GPU architecture.

[4]Contribution of this paper

3.1.1 Why can't I just replace all matrix multiplication with TensorCore and get higher performance? Unlike matrix-matrix multiplication, matrix factorization typically exhibits more dependency and less parallelism, and a more complicated memory access pattern. Therefore, matrix factorization cannot achieve the speed of matrix-matrix computation, but with algorithmic innovations called "blocked" or "tiled" QR could approach a significant fraction of peak performance. The idea of the tiling is to aggregate matrix-vector operations into fewer but bigger matrix-matrix multiplications, to increase arithmetic intensity (ratio between operations and number of elements) therefore enabling better data locality in the trailing matrix update. This technique is essential in bridging the gap between fast processor and slow memory, using the fast on-chip memories (registers, caches) to service most of the memory access. But because of the complex dependency, some part of the factorization is still matrix-vector and vector-vector based, which have much lower data locality. Modern algorithms and implementations usually divide each iteration of the factorization into two steps: ***panel*** *factorization* (slow, but small) and *trailing matrix* ***update*** (fast and big matrix multiplication), where most the floating point arithmetic is spent in the trailing matrix update. Based on this structure, a straight-forward strategy is to leave panel factorization intact, while replacing the trailing matrix update with TensorCore GEMM. However, this simple strategy is not enough to yield significant speedup for QR for the following reasons:

The matrix multiplication is not completely on the critical path. The MAGMA [14] QR uses a hybrid CPU/GPU architecture where panel factorization is on CPU, and trailing matrix update (GEMM) is on GPU. Due to algorithmic pipeline, the GEMM execution is completely or partially overlapped by the panel, thus speeding up GEMM has only limited effect on the overall QR speed. See Table 2 for an illustration. It seems to suggest that just by replacing the trailing matrix update with TC-GEMM leads to underwhelming speedups.

	Performance in TFLOPS					
block size	32	64	128	256	512	768
MAGMA QR	4.58	6.09	4.51	3.36	1.73	0.86
MAGMA QR with TC	4.63	7.02	4.87	3.52	1.64	0.86

Table 2: Performance MAGMA 2.5.2 SGEQRF() subroutine with trailing matrix using SGEMM vs. TC-GEMM on AMD Ryzen Threadripper 2970WX 24-Core Processor clocked at 2.4GHZ, with Titan V TensorCore, linked with MKL 2019.0.5. Matrix size 32768*16384

The tiled algorithm is not exposing enough data locality for significant acceleration The NVIDIA cuSolver SGEQRF subroutine is a pure GPU implementation of tiled Householder QR, which has GEMM on its critical path. However, the tiled QR algorithm does not provide enough data locality for the GPU and TensorCore. According to our benchmark in Table 3, TC-GEMM needs the smallest dimension to be at least 2048 to achieve maximum performance, while SGEMM needs around 1024. However, with such large block size, a large proportion of computation is spent in panel factorization which has low arithmetic intensity. To summarize, smaller block size B results in more FLOPs in matrix

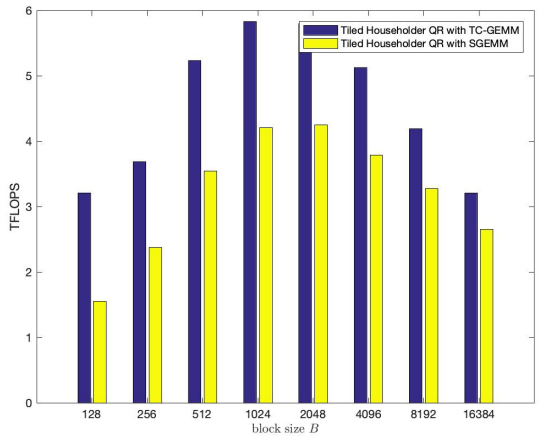


Figure 1: Estimated performance of blocked Householder QR with different block size B: trailing matrix update using TensorCore vs. without TensorCore, based on Table 3 and formula (4)

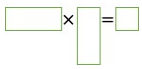
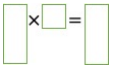
multiplication which runs slower, and bigger B results in fewer FLOPs in matrix multiplication, but they run faster. Typically, empirical tuning of B is required to achieve a good balance. The tiled Householder QR factorization spends 2 parts of computation in the panel (SGEQRF in Table 3), and n/B parts in the trailing matrix update (TC-GEMM/GEMM in Table 3); for details refer to this paper [4]. Thus we can estimate the performance of the conventional blocked Householder QR factorization with cuSOLVER SGEQRF() as panel and TC-GEMM/SGEMM as trailing matrix update:

$$S_{\text{HouseQR}}(B) = \frac{n/B + 2}{\frac{2}{S_{\text{SGEQRF}}(B)} + \frac{n/B}{S_{\text{GEMM}}(B)}} \tag{4}$$

which is plotted in Figure 1 for a 32768x16384 matrix. We can draw two conclusions: 1) In tiled Householder algorithms, enabling TensorCore for the trailing matrix update (TC-GEMM vs. GEMM) does increase the overall performance but only by around 30%; 2) Even with TC-GEMM enabled, the overall performance of the accelerated blocked Householder QR is no better than cuSOLVER SGEQRF(), which achieves >6 TFOPS.

3.1.2 Do more in less time: Recursive Gram-Schmidt QR Factorization There is another variant of QR algorithm that can also turn most of its operations into matrix-matrix multiplication—recursive QR. The idea of recursive QR has been explored by [17] to replace the **panel** factorization in QR. It's only used in the panel because it increases the number of operations needed to 2x that of Householder QR. The big increase in operation counts is probably the reason that recursive QR is not used often in practice. On the other hand, Recursive QR has the advantage of increased data locality, thus the limited use of QR in panel factorization is able to balance out its limited increased operation count and get a modest overall speedup.

In order to take better advantage of the TensorCore we will employ recursive QR as the overall QR algorithm. We mitigate the increase of operations, by resorting to a different basic QR algorithm—(modified) Gram-Schmidt (MGS)—rather than conventional Householder QR. It turns out that with MGS Recursive QR, the operation


k	TC-GEMM	SGEMM	TC-GEMM	SGEMM	SGEQRF
128	8.45	1.83	4.44	2.28	0.10
256	30.17	4.19	11.39	5.91	0.14
512	56.48	8.23	58.05	10.19	0.36
1024	72.39	12.43	77.58	12.80	0.79
2048	93.53	13.54	87.29	13.56	1.55
4096	97.82	12.31	92.72	12.81	2.71
8192	92.75	12.94	92.20	13.04	4.39
16384	82.32	12.96	83.40	13.12	6.67

Table 3: TC-GEMM and SGEMM Performance in TFLOPS as k changes from 128 to 16384 with fixed $m = 32768$. In columns 2-3, $A \in \mathbb{R}^{k\times m}$, $B \in \mathbb{R}^{m\times k}$. In columns 4-5, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times k}$. In column 6, $A \in \mathbb{R}^{m\times k}$. The device is an NVIDIA V100 PCIe version. The TC-GEMM, SGEMM, SGEQRF are from cuBLAS in the CUDA 10.1 distribution.

counts, which can be solved by recurrence (5), only increases moderately compared to Householder QR ($2mn^2$ vs $2mn^2 - \frac{2}{3}n^3$, at most 50% increase for $m \geq n$.) instead of a 2x increase. But because we can dramatically accelerate the matrix-matrix multiplication, it has the potential to result in faster overall execution time.

$$\begin{cases} T(n) = 2T(\frac{n}{2}) + mn^2, & n > 128 \\ T(n) = 2mn^2, & n = 128 \end{cases} \tag{5}$$

The basic idea of recursive QR is a quite simple one. 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 of $R_{12} = Q_1^T A_2$. 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 of the two halves can be recursed using this algorithm itself. The result of the original QR factors can be assembled like this:

$$[A_1|A_2] = [Q_1|Q_2] \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right] \tag{6}$$

Algorithm 1 Recursive Gram-Schmidt QR Factorization, with recursion cutoff size 128

```
1  function [Q,R] = RGSQRF(A)
2    [m,n] = size(A);
3    if n==128
4      [Q,R] = panelQR(A);
5      return
6    end
7    [Q1,R11] = RGSQRF(A(:,1:n/2));
8    R12 = Q1' * A(:,n/2+1:n);
9    [Q2,R22] = RGSQRF(A(:,n/2+1:n) - Q1 * R12);
10   Q = [Q1 Q2];
11   R = [R11 R12; zeros(n/2) R22];
12 end
```

The basic structure is the Algorithm 1, and the implementation uses cuSOLVER `SGEQRF()` as the `panelQR` (line 4) when the input

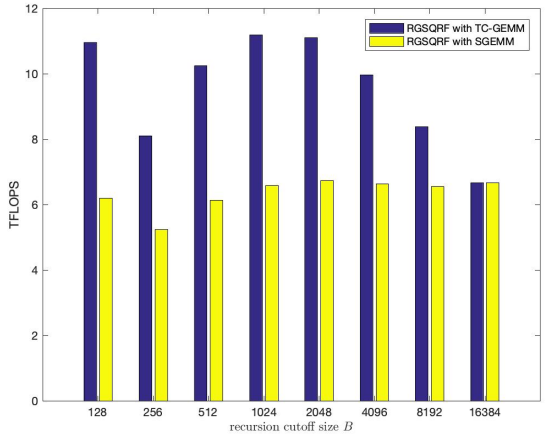


Figure 2: Estimated performance of RGSQRF with different block size B: matrix-multiplication using TensorCore (TC-GEMM) vs. without TensorCore (SGEMM), based on Table 3 and formula (7)

matrix A becomes small (e.g. $n = 128$). For matrix size $m \times n$, this algorithm roughly takes $2mn^2$ flops.

Let's do a similar estimate of the performance of this algorithm based on the benchmark in Table 3 for a 32768x16384 matrix and see if the RGSQRF algorithm can expose more data locality to perform better than tiled QR. In each function call `RGSQRF()`, roughly half of the flops is in matrix-matrix multiplication and the other half of the flops spent in the two recursion function calls. Thus, we have the following recursion for the performance (TFLOPS) of a $m \times n$ matrix:

$$S_{\text{RGSQRF}}(m,n) = \begin{cases} S_{\text{SGEQRF}}(m,n), & n = B \\ 2/(\frac{1}{S_{\text{RGSQRF}}(m,n/2)} + \frac{1}{S_{\text{TC-GEMM}}(m,n/2)}), & n > B \end{cases} \tag{7}$$

We calculate $S_{\text{RGSQRF}}(32768, 16384)$ using a range of cutoff B and show the result in Figure 2. Compare this figure with Figure 1, it's clear that Recursive QR exposes more data locality than tiled QR, and allows better speedups from TensorCore. Factoring the extra 20% computations, the RGSQRF still gains significant speedups from TC-GEMM, and surpasses the performance of SGEQRF() by about 37%. Furthermore, Recursive QR can achieve optimal performance even at a small panel size $B = 128$, which would be important for reducing the footprint and global memory traffic in the next subsection.

3.1.3 Do more in less time: Communication Avoiding Gram-Schmidt QR Factorization Panel The recursive Gram-Schmidt QR seems to be able to benefit from TensorCore substantially, but the unaccelerated panel factorization (cuSOLVER SGEQRF()) limits the overall speedup severely. In this subsection, we once again trade computations for more data locality (fewer data movements/communication) in the panel, by employing the communication avoiding Gram-Schmidt QR.

The challenge in fast `panelQR` is that of data locality and parallelism. The conventional Householder panel has sequentially dependent iterations, and the working-set is the whole panel which cannot fit in fast memory on GPU (register files+ shared memory). Fortunately for QR, there's a communication avoiding QR (CAQR) [1] variant that simultaneously improve parallelism and data locality, at the cost of more computations. Our `panelQR` is based on CAQR, with the Modified Gram Schmidt QR replacing Householder QR used in [1]. The idea of CAQR can be illustrated in equation (8).

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \overset{①}{=} \begin{bmatrix} Q_{11}R_1 \\ Q_{12}R_2 \\ Q_{13}R_3 \\ Q_{14}R_4 \end{bmatrix} \overset{②}{=} \begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}$$
$$\overset{③}{=} \begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix} \begin{bmatrix} Q_{21} \\ Q_{22} \\ Q_{23} \\ Q_{24} \end{bmatrix} R \tag{8}$$
$$\overset{④}{=} \begin{bmatrix} Q_{11}Q_{21} \\ Q_{12}Q_{22} \\ Q_{13}Q_{23} \\ Q_{14}Q_{24} \end{bmatrix} R \overset{⑤}{=} QR$$

In (8), there are 5 steps indicated by the number over the equality sign. In the ① step, we divide a tall matrix A evenly into 4 smaller matrices (still tall, more rows than columns), and QR factorize them independently. In step ② 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 ③, we factorize the vertically stacked Rs (potentially carry this process recursively). In ④, we do 4 matrix-matrix multiplications for the 4 corresponding Q factors. In ⑤ we reinterpret the result as the QR factors of original A. The reason Q is orthogonal, is that in step ④ the 4 matrix-matrix multiplication is equivalent to the product of two orthogonal matrices (second line) and therefore is orthogonal.

Practically, we fix our panel to be of 32 columns with m rows and decompose the matrix A into 256x32 submatrices (step ①). On V100 GPU, the 256x32 submatrix can fit into shared memory so that we only need to read and write global memory once. These 256x32 blocks are independently factorized using the modified Gram-Schmidt algorithm into QR factors; see algorithm 2. To map this algorithm to GPU effectively, we let each threadblock QR factorize one 256x32 block. We launch 256 threads, with each threads reading, processing, and writing a single row of the 256x32 block. The most time-consuming part is line 7 where reductions are needed (vector inner products across threads). We use CUB template library[5] from NVIDIA Research to have a threadblock level fast reduction. We manually unroll the loop 4 ways to expose more instruction-level parallelism, and to reduce the number of reductions by a factor of 4. In step ④ we use cuBLAS batched `SGEMM()` subroutine to do the matrix multiplications in parallel. We recurse in step ③, until the number of rows is below 256 so that a single threadblock will suffice.

In summary, our CAQR implementation has two salient features: 1) the Gram-Schmidt process is run completely within shared memory; 2) all the inter-threadblock communication/synchronization happens in the batched SGEMM() which is extremely fast. Hence our CAQR panel reads global memory minimally ($\log_8(m/256)$

[5] https://nvlabs.github.io/cub/

passes to the panel) , and have minimal cross threadblock synchronization and communication.

Algorithm 2 256x32 Modified Gram-Schmidt QR, as implemented in the CUDA kernel

```
function [Q,R] = mgs(A)
  [m,n] = size(A);
  Q = A; R = zeros(n);
  for k=1:n
    R(k,k) = norm(Q(:,k));
    Q(:,k) = Q(:,k)/R(k,k);
    R(k,k+1:n) = Q(:,k)' * Q(:,k+1:n);
    Q(:,k+1:n) = Q(:,k+1:n) - Q(:,k) * R(k,k+1:n)
  end
end
```

For a 32768x128 panel, our hand coded CAQR panel achieves 0.33TFLOPS, which is 3.3x faster than cuSOLVER SGEQRF(). As a result, the RGSQRF() algorithm equipped with the new CAQR panel is estimated to achieve 27 TFLOPS on 32768x16384 matrix using the estimation formula (7). We will have more comprehensive empirical study in section 4; in fact as data in Figure 6 shows our actual implementation achieves 26.2TFLOPs, which is very close to what we have estimated.

3.2 Linear Least Square Problem with QR Factorization

One important use of QR factorization is to solve linear least square problems.

3.2.1 Numerical Issues A natural concern for using the half precision TensorCore matrix-matrix multiplication is the potential loss of accuracy and stability. In the case of QR, two kinds of accuracy are of importance: the backward error and the orthogonality of the Q factor. The backward error is $\frac{||A-\hat{Q}\hat{R}||}{||A||}$ and the orthogonality of $\hat{Q}$ is $||I - \hat{Q}^T\hat{Q}||$.

Ideally, these two numerical errors should be zero, but because of roundoff errors (exacerbated by using low precision TensorCore) and potential loss of orthogonality of Gram-Schmidt QR, the QR factorization might not be accurate enough for solving the LLS problem.

3.2.2 Direct Solve with QR The accuracy of the direct solution of LLS problem using QR factorization using (3) depend on the accuracy of the *QR* factorization. To measure the accuracy of a solution to the linear least square problem $\min_x ||Ax - b||$, we use the following accuracy metric:

$$A^T(A\hat{x} - b)$$

for a computed solution $\hat{x}$. Ideally, this metric should be 0, but will not be exactly zero due to roundoff errors in the QR factorization. Therefore smaller is better for this accuracy test for LLS.

3.2.3 Iterative Refinement It can be seen that directly solve the LLS problem with our low precision QR factorization may not lead to sufficient accuracy. To achieve higher accuracy, we refine the solution. There are two approaches to this task. One is actually called iterative refinement in the literature [5, 6, 12, 25]. Another one, which appears to be new for this purpose is what we are going to introduce. It's a Krylov subspace iterative solver for LLS, coupled with our low-precision QR factorization as **preconditioner** to achieve high accuracy and fast convergence. This idea blurs the distinction between direct solver and iterative solver; it inherits the stability and robustness of direct solver, while retains the flexibility and the iterative nature of Krylov iterative solver. We use the CGLS iterative solver, which is mathematically equivalent to Conjugate Gradient on the normal equation, but numerically more stable. We list the algorithm with the QR factorization in Algorithm 3.

Algorithm 3 LLS High Accuracy Solver: CGLS with RGSQRF as Preconditioner[a]

```
function [x] = cgls_qr(A,b)
  [Q,R] = RGSQRF(A); % TensorCore QR
  [m,n] = size(A);
  x = zeros(n,1);
  r = b - A*x;
  s = A'*r;
  p = s;
  norms0 = norm(s);
  gamma  = norms0^2;
  for k=1,2,...
    q      = A*(R\p);
    delta  = norm(q)^2;
    alpha  = gamma/delta;
    x      = x + alpha*p;
    r      = r - alpha*q;
    s      = R\(A'*r);
    norms  = norm(s);
    gamma1 = gamma;
    gamma  = norms^2;
    beta   = gamma / gamma1;
    p      = s + beta*p;
  end
end
```

[a] The convergence test is omitted. This presentation is adapted from Per Christian Hansen and Michael Saunders at https://web.stanford.edu/group/SOL/software/cgls/matlab/cgls.m

This algorithm first calls upon the fast RGSQRF to do QR factorization, and then runs CGLS algorithm, with the R factor as the right preconditioner for A. For a sufficiently accurate QR factor R, AR^{-1} should be fairly well-conditioned, which means that $\kappa(AR^{-1})$ is small (close to 1, ideally). The convergence rate is linear; specifically the error is reduced by at least a constant factor in every iteration [39]:

$$e_k = e_0 \left(\frac{\kappa(AR^{-1}) - 1}{\kappa(AR^{-1}) + 1} \right)^k$$

With perfect QR factorization $\kappa(AR^{-1}) = \kappa(Q) = 1$, and CGLS converges in 1 iteration. With imperfect QR, we need slightly more iterations to converge; see experiment section 4.2 for some empirical examples.

3.3 Orthogonalization

Another important application of QR factorization is called orthogonalization— finding an orthogonal basis of the columns of matrix A (which is given by Q in the QR factorization). The RGSQRF

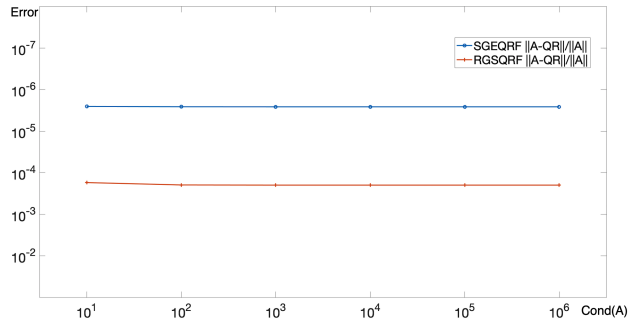


Figure 3: QR factorization accuracy; backward error (smaller is better) RGSQRF vs. SGEQRF. matrix size 32768*16384, SVD arithmetic distribution

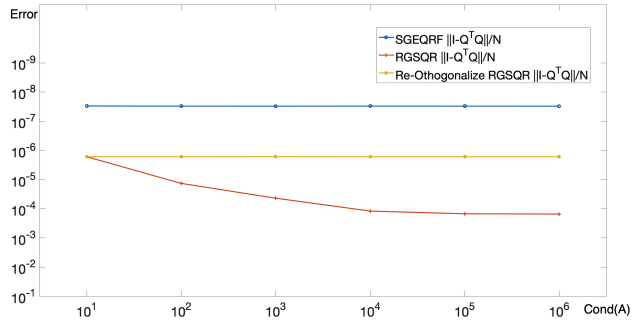


Figure 4: QR factorization orthogonality accuracy: SGEQRF vs. RGSQRF vs. RGSQRF-Reortho, matrix size 32768*16384, SVD arithmetic distribution

algorithm inherits one of the deficiencies of Gram-Schmidt orthogonalization procedure in the loss of orthogonality of the Q factor for ill-conditioned matrices. If we need accurate orthogonalization results for some application then we need to refine the orthogonality of the Q factor. To do that we can re-orthogonalize the Q factor again using the RGSQRF subroutine: $Q = Q_2 * R_2$. This procedure is akin to the block re-orthogonalization in Gram-Schmidt QR [38]. The general result can be summarized by "twice is enough": orthogonalizing twice should bring the orthogonality to fully working precision. For some analysis behind this technique, we refer to [18]; here we show some empirical study that demonstrates the practical efficacy of re-orthogonalization. From Figure 3 we see that the backward error ($||A-QR||/||A||$) is not affected by cond(A) for both RGSQRF and SGEQRF, and it's up to the full working precision, half and single respectively. From Figure 4 we see that SGEQRF is fully orthogonal up to working precision, whereas RGSQRF's orthogonality deteriorates with the increase of cond(A). However, with re-orthogonalization, RGSQRF-ReOrtho can be brought back to orthogonality up to working precision. **With re-orthogonalization, RGSQRF is 3.7x to 7.7x faster than SGEQRF+SORMQR from Figure 5.**

3.4 Optimal Low Rank Approximation

Yet another application of QR factorization is computing optimal low rank approximation of a tall-skinny (data) matrix using the

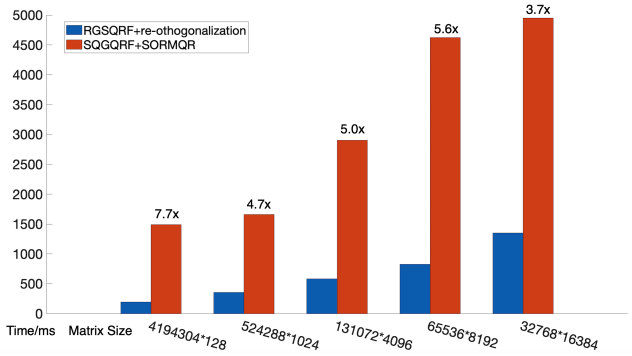


Figure 5: QR factorization performance: RGSQRF-Reortho (left bar) vs. cuSolver SGEQRF plus generating Q (SORMQR) (right bar) for different matrix sizes.

	$\|\|A - QU_r\Sigma_r V_r^T\|\|/\|\|A\|\|$	
rank r	RGSQRF-SVD	SGEQRF-SVD
16	9.77e-01	9.77e-01
64	9.08e-01	9.08e-01
128	8.18e-01	8.18e-01
256	6.49e-01	6.49e-01
512	3.53e-01	3.53e-01
	QR-SVD	
	RGSQRF-SVD	SGEQRF-SVD
time(ms)	274.95	1755.19

Table 4: QR-SVD based optimal low rank approximation for a tall-skinny matrix of size 524288*1024. The matrix is randomly generated with arithmetic singular value distributions, cond(A) = 10^6.

efficient QR-SVD algorithm: $A = QR$, $R = U\Sigma V^T$, and we have SVD of $A = QU\Sigma V^T$. The optimal low rank approximation has many uses such as data compression, dimension reduction, principal component analysis, etc. For tall-skinny matrix, the dominant computations are spent in the first step—QR factorization, which can be substantially accelerated using our proposed RGSQRF. With SVD, we can truncate it to low rank to obtain the optimal rank-r approximation $A \approx A_r = QU_r\Sigma_r V_r^T$. Since truncation error is most likely to be dominant, the roundoff errors in RGSQRF is insignificant, thus no refinement or re-orthogonalization is needed. **See Table 4; we get the same quality optimal low rank approximation as single precision QR-SVD, while 6.4x faster.**

3.5 Rescaling the matrix to mitigate over/underflow

One of the important haphazard in using TensorCore is the limited range of TensorCore internal half precision floating point format. For example, if a matrix element is larger than 65504 then when it gets converted to half precision in TensorCore there's an overflow that either result in infinity or unbounded error. To avoid such catastrophe, we can take advantage of one property of QR factorization: rescaling the columns of the matrix A does not affect the QR factorization. Specifically, we can arbitrarily scale the columns of

A (equivalent to multiplying a diagonal matrix P to the right of A: AP), and The Q factor remains unchanged; the columns of R factor will get scaled by the same P: $AP = QRP$ so that we can recover the QR factorization of the original matrix easily. This property holds not only mathematically but also numerically, as long as the scaling itself is accurate. This property allows us to rescale a badly-scaled matrix A to the range of half precision floating point: 6×10^{-8} to 65504, such that overflow will never happen, and underflow will be reduced or eliminated. This scaling can be done fully automatically and very cheaply by examining every column of A for a scaling factor. Since orthogonal transformation preserves 2-norm, once the initial matrix is properly scaled then all intermediate operations will not overflow. Note that on the contrary, LU factorization does not guarantee this.

3.6 Error Analysis

This section is devoted to the error analysis of the RGSQRF algorithm. Basically, two kinds of error would be considered: the backward error and the orthogonality of the Q factor. For calculating the backward error, we can use the below equation.

$$\frac{||A - \hat{Q}\hat{R}||_2}{||A||_2}$$

In [40] author has found the bounding of backward error $\frac{||A-\hat{Q}\hat{R}||_2}{||A||_2} \leq cv$ in which c is a constant number and v is unit roundoff. It is obvious that in this equation the backward error is small and independent from the condition number. As discussed in the previous section, the orthogonality error can be defined by

$$||I - \hat{Q}^T\hat{Q}||_2$$

Based on our experiment, this error is between Recursive-CGS and Recursive-MGS. In [18] the authors proved that for CGS the loss of orthogonality can be bounded in terms of the square of the condition number $\kappa(A)$. In contrast, Bjorck [40] proved that the loss of orthogonality in MGS depends only linearly on $\kappa(A)$. So, we can claim that the error bound of our method would be between bounds of CGS and MGS, which in worse case is the square of the condition number $\kappa(A)^2$ times the unit roundoff. However, according to our experimental result, it is closer to ϵ times $\kappa(A)$. Moreover, if we used re-orthogonalization in our method, the error bound for orthogonality is $||I - \hat{Q}^T\hat{Q}||_2 \leq \epsilon$ which removes its dependence on the condition number of A. In this way, we can conclude that re-orthogonalization will decrease the amount of the error bound and two times orthogonalization are enough for ensuring the orthogonality of matrix Q [3, 18].

4 Experiments

In this section, we conduct a comprehensive empirical study on the numerical behavior (accuracy), and performance behavior of our proposed RGSQRF and Linear Least Square Solver. We pay special attention to how enabling/disabling TensorCore for matrix multiplication affects the performance.

For all the experiments we use a Redhat 7 Linux workstation with NVIDIA V100 (PCIe version) GPU. The CUDA version is 10.1, which contains a C++ compiler and libraries cuBLAS and cuSOLVER. For the LLS experiments, we used random matrix generation routine

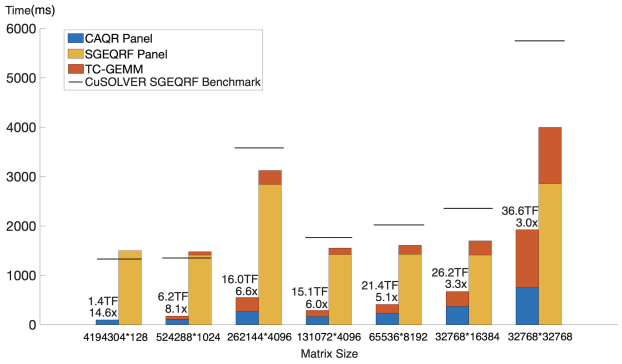


Figure 6: RGSQRF performance with different panel: CAQR panel (left bar) vs. SGEQRF panel (right bar). In the label TF means TFLOPS, dx means d times speedup over baseline cuSOLVER SGEQRF

from MAGMA 2.5.1 to generate a random matrix with a specific condition number and singular value distribution.

4.1 QR Factorization

4.1.1 The effect of the CAQR panel on the performance of RGSQRF Figure 6 illustrates the performance of our two attempts and the comparison with cuSOLVER SGEQRF. As we can see that for a large scale matrix, the speedup of TensorCore accelerated RGSQRF is between 3.0x to 14.6x, depending on the shape of the matrix. From Figure 6, CAQR panel contributes more when the matrix is skinny, whereas TC-GEMM contributes more when the matrix is squarish. In summary, it's clear that the CAQR panel is essential in the RGSQRF performance improvement over SGEQRF by comparing the left bar and right bar in Figure 6.

4.1.2 The effect of TensorCore on performance in panel and trailing matrix update We did some extra experiments to show the effect of enabling/disabling TensorCore in both the *panel* and *update*, shown in Figure 7. The first uses TensorCore for both panel/update. The second bar uses TensorCore only in update, the best one that's reported in Figure 6. The third bar disables TensorCore.

The first thing we notice by comparing the left bar with the middle bar is that TensorCore does not help much in the panel. Therefore, we decide not to use TensorCore in the CAQR panel to avoid losing accuracy for little gain in speed. On the other hand, comparing the middle bar to right bar indicates that TensorCore in update is critical and worth using. In fact, without TensorCore, RGSQRF may speed down compared to cuSOLVER SGEQRF, especially for squarish matrices. In summary, TensorCore contributes critically to the speedups we see in RGSQRF. Note that, the highest TFLOPS of RGSQRF is 36.6 (32768*32768), which utilizes around 37.4% of the TensorCore peak performance (97.82 TFLOPS).

4.2 Linear Least Square Problem

Unlike QR, whose accuracy only depends on condition number, to refine LLS solution the CGLS iterative solver performance depends on the singular value distribution of A. To cover a comprehensive variety of different singular value distribution and condition number, we use the following randomly generated matrix. 1) each element is i.i.d. from uniformly distributed random number within

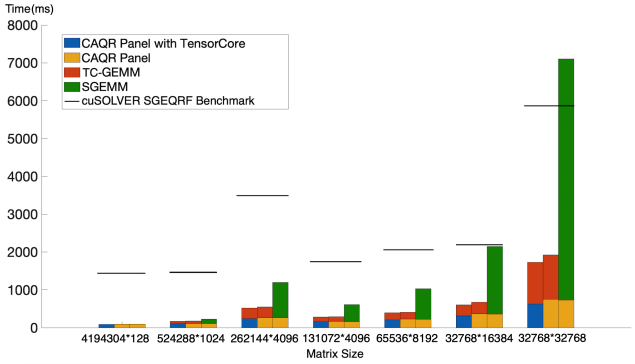


Figure 7: Performance of RGSQRF with TensorCore enabled vs. disabled in panel and trailing matrix update. TensorCore in panel/update are (on,on) for left bar, (off, on) for middle bar, and (off, off) for right bar.

(0,1) and (-1,1); 2) each element is i.i.d from a normally distributed random number with mean 0 and standard deviation 1; 3) random matrix with specified condition number and geometric singular values (σ_i) distribution: $[\log \sigma_1, \ldots, \log \sigma_n]$ are evenly spaced; 4) random matrix with specified condition number and arithmetic singular values (σ_i) distribution: $[\sigma_1, \ldots, \sigma_n]$ are evenly spaced; 5) random matrix with clustered singular values: all but the smallest singular values are 1.

4.2.1 Performance Based on the performance of QR factorization, we are also expecting a considerable speedup on solving LLS problems. To get the same accuracy level with direct LLS solver provided by cuSOLVER, we combine RGSQRF and CGLS (Algorithm 3) to refine the solution accuracy. Figure 8 shows the comparison between the time cost of RGSQRF plus CGLS iterative solver and direct solver (SGEQRF+SORMQR+STRSM), note that the RGSQRF solution is able to attain **double precision accuracy**. Obviously, we spend extra time in CGLS when compared with direct solvers, which results in somehow a lower speedup than QR factorization. But it is still a tremendous improvement in solving LLS problems. Similarly, there is some tendency that taller and thinner matrices tend to perform better, which is in line with our observation from the experiments on QR factorization.

Generally speaking, CGLS converges rapidly with preconditioned AR^{-1}. In the case of uniformly random matrix 32768×16384, it can reach a competitive accuracy in 20 iterations.

However, the uniform matrix is typically well-conditioned and it should have a fast converge speed. The convergence rate of an iterative solver like CGLS depends strongly on the spectrum property of the matrix A. To make the LLS study more general, we generate different types of matrices with different singular value distribution and condition number. We expect results to be condition-distribution-related, that is, the larger condition number the matrix has, the more iterations it will take. In some extreme cases, CGLS cannot converge to the highest accuracy and we will discuss it in more detail next subsection. Figure 8a to Figure 8h illustrate the relationships in terms of condition number, distribution and number of iterations, and it is consistent to our anticipation. **We observe that in almost all cases RGSQRF with refinement outperforms single/double precision direct LLS solver using [S/D]GEQRF by up to 8.9x/13.5x while achieving the same accuracy, respectively.**

4.2.2 Accuracy At first we would like to show the observations on the accuracy based on $x = R^{-1}(Q^T b)$. Because RGSQRF involves half precision, so we are not expecting to see as accurate results as cuSOLVER can provide. As the accuracy showed in Fig 9, we can conclude that in most cases, RGSQRF direct solver performs worse than SGEQRF solver and the difference is around two orders of magnitude. It explains why we need iterative methods as a safeguard.

Fig 9 also compares DGEQRF direct solver, SGEQRF direct solver and RGSQRF iterative solver accuracy with several condition numbers. For RGEQRF iterative solver, we choose a somehow best tolerance that will give us a relatively accurate result and reasonable converge speed. We can observe that if the matrix condition is not very bad, RGSQRF and CGLS can generate at least the same level of accuracy with DGEQRF direct solver with a small number of iterations(shown by the digits in Fig.9).

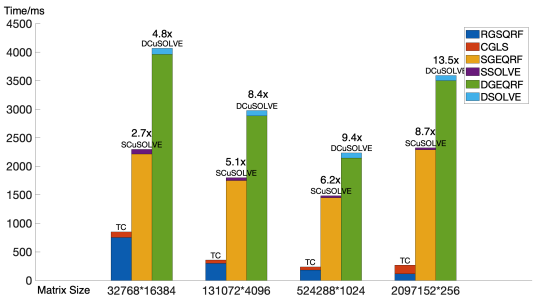
To sum up, in terms of accuracy, we can claim that RGSQRF with CGLS refinement can provide a reliable result when compared with single/double precision Householder QR LLS direct solver at much faster speed.

According to the experiments on SVD geometric distribution (Fig 8d), we can find the performance on this type of matrix is not as impressive as other types. The reason is that CGLS takes 20-30 iterations to converge to 10^{-12} (the same accuracy with DCuSOLVE), while other matrix types typically take less than 10 iterations to converge. We also test SVD geometric distribution with $cond = 10^4$ and it reveals that for matrix size 32768*16384, it needs 200 iterations—which is the max number of iteration we can tolerate—to converge to 10^{-6} and it's because of the very difficult distribution of singular values. This represents a stress case for our refinement procedure. We are still able to achieve single precision accuracy with around 2x speedup, however, we can not achieve double precision accuracy profitably. If high accuracy is needed double precision QR direct solve should be used instead, which is beyond the capability of single precision QR, and certainly beyond the mixed half precision RGSQRF with refinement.

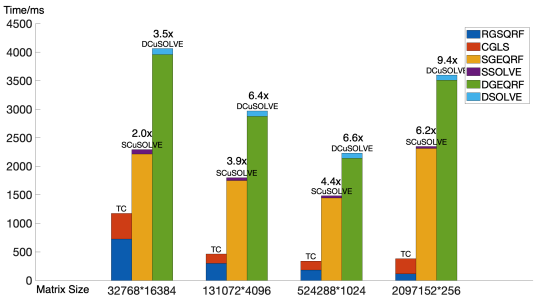
5 Related Work

NVIDIA introduced TensorCore technology with its Volta architecture [32] in 2017. Resources about NVIDIA TensorCore include detailed micro-architecture analysis and benchmarking [29], an early report on the programmability, performance, and precision [31]. In [11] important parallel primitives reduction and scan is accelerated with TensorCore. In [21–23] TensorCore was used for accelerating linear system solvers in the framework of hybrid CPU/GPU linear algebra package MAGMA [14]. There are numerous use cases of half precision or even lower precision in the application of neural networks.

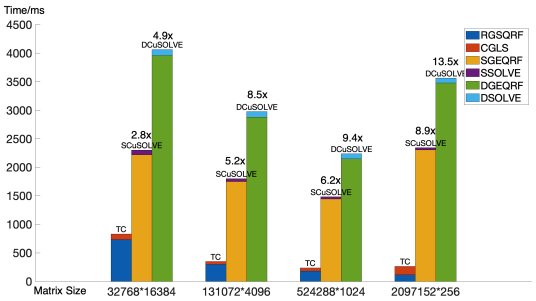
The QR factorization, along with LU and Cholesky factorization form the one half of important matrix factorizations in numerical linear algebra. QR factorization can be used to solve linear system, LLS problems, orthogonalization of a set of vectors, and eigendecompositions; see the encyclopedic book [19] for more



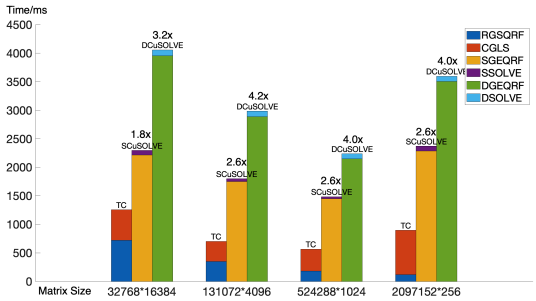
(a) **Matrix of Type 1: Random Uniform on (0, 1)**



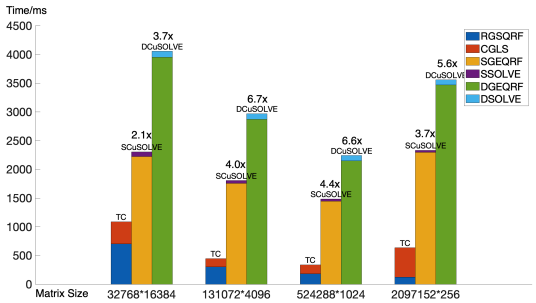
(b) **Matrix of Type 2: random uniform on (-1, 1)**



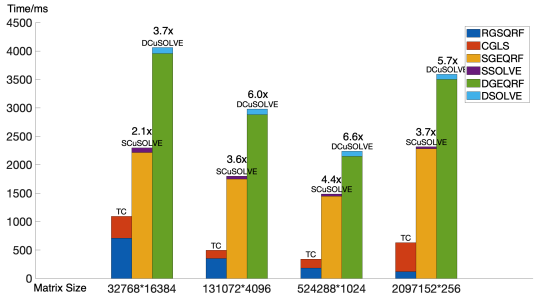
(c) **Matrix of Type 3: random normal with mean=0, standard deviation=1**



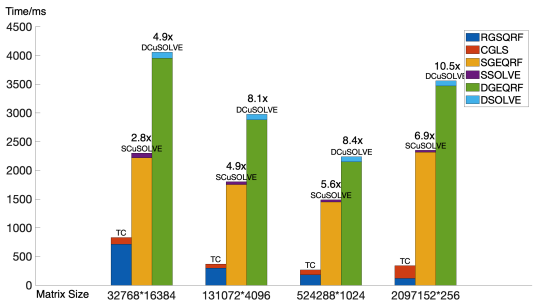
(d) **Matrix of Type 4: SVD geometric distribution with** $cond = 10^3$



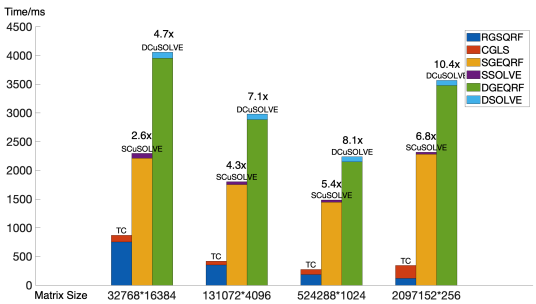
(e) **Matrix of Type 5: SVD arithmetic distribution with** $cond = 10^5$



(f) **Matrix of Type 6: SVD arithmetic distribution with** $cond = 10^6$



(g) **Matrix of Type 7: SVD cluster distribution with** $cond = 10^5$ **and** $\sigma_i = \{1, ..., 1, \frac{1}{cond}\}$



(h) **Matrix of Type 8: SVD cluster distribution with** $cond = 10^6$ **and** $\sigma_i = \{1, ..., 1, \frac{1}{cond}\}$

Figure 8: Performance in milliseconds and speedups of three linear square problem solvers: RGSQRF iterative solver(left bar), cuSolver SGEQRF direct solver(middle bar) and cuSolver DGEQRF direct solver(right bar) for different matrix types and sizes.

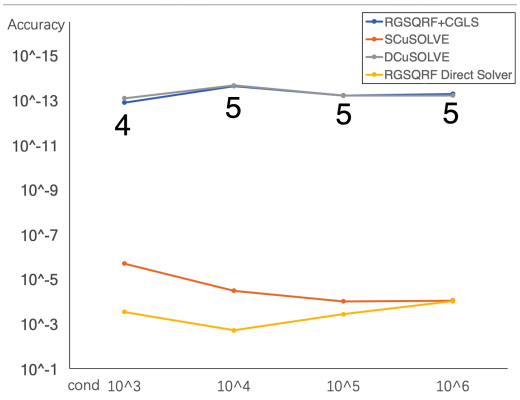


Figure 9: LLS accuracy: matrix size 32768*16384 with SVD cluster2 distribution, condition number varies from 10^3 to 10^6. Comparison between SCuSOLVE, DCuSOLVE,RGSQRF Direct Solver and RGSQRF+CGLS. The numbers along with the lines indicate number of CGLS refinement iterations.

details and pointers. These factorizations for the core of popular linear algebra packages such as LAPACK [1] and Eigen [20] for general CPUs, PLASMA [15] on multi-core systems, ScaLAPACK [7] and Elemental [35] for distributed memory systems, and cuSOLVER[6]/cuBLAS[7] for NVIDIA GPU accelerators as part of CUDA libraries, and SLATE [30] on distributed heterogeneous CPU/GPU systems. There are primarily three main algorithms for QR factorization: classic Gram-Schmidt, modified Gram-Schmidt, and Householder QR [27]. See a blog post from Cleve Moler[8] for a simple comparison, and the book [37] for details. The high performance implementation of Householder QR depends on blocking, i.e. aggregating several Householder reflections into a single matrix-matrix multiplication. The scheme was developed in [36] and used in virtually all high performance numerical linear algebra packages. Communication-Avoiding QR is discussed in [2, 13].

The use of QR factorization as a stable method to solve the LLS problem is a standard direct method. Iterative methods for the LLS problems are also possible, and maybe preferred for very large scale and sparse problems. CGLS appeared in [24] together with the discovery of the Conjugate Gradient method; there's another mathematically equivalent but numerically more stable one called LSQR [34]. In this paper, we take a somewhat unusual approach in using the iterative method for a general dense problem.

The roundoff error analysis of half precision floating point arithmetic is only emerging. The report [26] provides some statistical roundoff error analysis that is more suitable for half precision, as the traditional deterministic analysis is too pessimistic to give any useful error bound. The report [8] provides error analysis for the fused multiply-addition matrix multiplication on TensorCore. These papers [9, 10] proposes and analyzes a mixed half, single, and double precision linear solver.

The closest related work is probably the linear solver based on TensorCore [21–23]. This work shares some ideas with those recent works in that both compensate the loss of precision from TensorCore by combining an iterative solver or iterative refinement. Both contribute to the broad effort in bringing TensorCore to linear algebra. The distinction is that this paper considers QR factorization instead of LU factorization, and proposes a GPU only instead of hybrid CPU/GPU.

There is another related work of mixed precision (single/double) QR factorization [28]. To perform most of computation using BLAS-3 kernels, they do a GEMM at first and factorize A^TA to obtain R. However, the orthogonality error of CholQR depends quadratically on the condition number of the input matrix. The authors propose a method that some of the intermediate results are accumulated in the doubled precision and they prove that with this method, the orthogonality error is only bounded by $\kappa(A)$. In contrast, our method doesn't seem to double the condition number of the input matrix.

A recent paper also utilizes TensorCore to perform QR factorization and they achieve 3x faster than CuSolver [33]. But they are limited in that only very tall and skinny matrices (with 16 columns) can be factorized faster, while our method is not only faster but also can handle arbitrary shapes.

6 Conclusions and Future Work

Neural engines are characterized by extreme performance and low precision/range. We explored accelerating QR factorization using neural engines accurately and reliably. First, we demonstrate that simply replacing matrix-multiplication with neural engines in conventional QR does not result in much speedup, due to inadequate data locality and parallelism. We then devised a novel recursive QR algorithm and a new panel that expose a much higher degree of data locality at the cost of increased computation. The tradeoff is essential in effectively exploiting the TensorCore on V100 GPU which results in 3.0x-14.6x speedup over cuSOLVER SGEQRF.

On the other hand, the low precision of the neural engine may cause a loss of accuracy. To regain accuracy, depending on how QR is used we may have different safeguarding procedures. For the least square solve, we proposed a novel Krylov subspace method as iterative refinement. For orthogonalization, we propose re-orthogonalization to bring orthogonality to full working precision. For truncated QR-SVD we often don't need any refinement as the numerical error is dwarfed by the truncation error.

We conclude by saying that neural engines can significantly accelerate matrix computations, but intricate analysis and engineering are needed to balance the time/space complexity of algorithm, utilization of hardware resources, and numerical stability and accuracy. One interesting question is how this particular effort in QR informs the use of neural engines in other matrix computations. Firstly, with neural engines, the already big gap between fast compute and slow data movement becomes one order of magnitude bigger. This disrupted balance means that communication avoiding algorithms (often recursive ones) are essential on neural engines. Such algorithms while theoretically superior, used to be only profitable for special cases (tall and skinny, small matrix on many processors, etc). Secondly, careful optimization of the non neural engine accelerated operations become more critical because neural engine is simply *so much faster*, that the speedup is bottlenecked by the unaccelerated parts. Lastly, more research and developments to expose more ways to use neural engines beside the matrix multiplication interface

[6]https://developer.nvidia.com/cusolver
[7]https://developer.nvidia.com/cublas
[8]https://blogs.mathworks.com/cleve/2016/10/03/householder-reflections-and-the-qr-decomposition/

will substantially increase the applicability and efficacy of neural engine for much more applications.

Acknowledgements

We are very grateful for the valuable suggestions from the anonymous reviewers and their time and efforts in reviewing the manuscript. This work has benefited greatly from correspondence with Prof. Åke Björck (University of Linköping) and Prof. Julien Langou (University of Colorado Denver), and all the errors are our own.

We would like to thank the University of Houston and the state of Texas for providing the startup funding in supporting this work. We acknowledge the use of the Sabine Cluster and the advanced support from the Research Computing Data Core at the University of Houston to carry out the research presented here. We are also grateful for the generous of hardware donation of NVIDIA Corporation. We gratefully acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604

[2] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. 2011. Communication-Avoiding QR Decomposition for GPUs. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, Anchorage, AK, USA, 48–58. https://doi.org/10.1109/IPDPS.2011.15

[3] Jesse L. Barlow and Alicja Smoktunowicz. 2011. Reorthogonalized Block Classical Gram–Schmidt. *arXiv:1108.4209 [math]* (Aug. 2011). http://arxiv.org/abs/1108.4209 arXiv: 1108.4209.

[4] Christian Bischof and Charles Van Loan. 1987. The WY Representation for Products of Householder Matrices. *SIAM J. Sci. Statist. Comput.* 8, 1 (Jan. 1987), s2–s13. https://doi.org/10.1137/0908009

[5] Åke Björck. 1967. Iterative refinement of linear least squares solutions I. *BIT* 7, 4 (Dec. 1967), 257–278. https://doi.org/10.1007/BF01939321

[6] Åke Björck. 1968. Iterative refinement of linear least squares solutions II. *BIT* 8, 1 (March 1968), 8–30. https://doi.org/10.1007/BF01939974

[7] L. S. Blackford, Jaeyoung Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, and Jack Dongarra. 1997. *ScaLAPACK user's guide*. SIAM.

[8] Pierre Blanchard, Nicholas J Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. 2019. *MIXED PRECISION BLOCK FUSED MULTIPLY-ADD: ERROR ANALYSIS AND APPLICATION TO GPU TENSOR CORES*. Technical Report. The University of Manchester. 16 pages.

[9] Erin Carson and Nicholas J Higham. 2017. *Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions*. Technical Report. University of Manchester. http://eprints.maths.manchester.ac.uk/

[10] Erin Carson and Nicholas J Higham. 2017. A New Analysis of Iterative Refinement and its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *SIAM Journal on Scientific Computing* 39, 6 (2017), A2834–A2856. https://doi.org/10.1137/17M1122918

[11] Abdul Dakkak, Cheng Li, Isaac Gelado, Jinjun Xiong, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. *Proceedings of the ACM International Conference on Supercomputing - ICS '19* (2019), 46–57. https://doi.org/10.1145/3330345.3331057 arXiv: 1811.09736.

[12] James Demmel. 2007. *Extra-precise Iterative Refinement for Overdetermined Least Squares Problems*. Technical Report. Lapack Working Notes.

[13] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing* 34, 1 (Jan. 2012), A206–A239. https://doi.org/10.1137/080731992

[14] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating numerical dense linear algebra calculations with GPUs. *Numerical Computations with GPUs* (2014), 3–28. https://doi.org/10.1007/978-3-319-06548-9_1 ISBN: 9783319065489.

[15] Jack Dongarra, Piotr Luszczek, and David Stevens. 2017. *PLASMA 17 Performance Report Linear Systems and Least Squares*. Technical Report. University of Tennessee, LAPACK Working Note #292.

[16] Iain S Duff and Serge Gratton. 2006. The Parallel Algorithms Team at CERFACS.

[17] E. Elmroth and F. G. Gustavson. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development* 44, 4 (July 2000), 605–624. https://doi.org/10.1147/rd.444.0605

[18] Luc Giraud, Julien Langou, Miroslav Rozložník, and Jasper van den Eshof. 2005. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numer. Math.* 101, 1 (July 2005), 87–100. https://doi.org/10.1007/s00211-005-0615-4

[19] Gene H. Golub and Charles F. Van Loan. 2012. *Matrix Computations*. JHU Press. https://books.google.com/books?id=5U-l8U3P-VUC

[20] Gaël Guennebaud, Beno^it Jacob, and others. 2010. Eigen v3. (2010). http://eigen.tuxfamily.org

[21] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack J. Dongarra. 2018. The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques. In *Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I*. 586–600. https://doi.org/10.1007/978-3-319-93698-7_45

[22] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC*.

[23] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. 2017. Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers. In *8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*.

[24] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. *Methods of conjugate gradients for solving linear systems*. Vol. 49. NBS Washington, DC.

[25] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027

[26] Nicholas J. Higham and Theo Mary. 2018. *A New Approach to Probabilistic Rounding Error Analysis*. Technical Report. The University of Manchester.

[27] Alston S. Householder. 1958. Unitary Triangularization of a Nonsymmetric Matrix. *J. ACM* 5, 4 (Oct. 1958), 339–342. https://doi.org/10.1145/320941.320947

[28] Yamazaki Ichitaro., Tomov Stanimire., and Dongarra Jack. 2015. Mixed-Precision Cholesky QR Factorization and Its Case Studies on Multicore CPU with Multiple GPUs. *SIAM Journal on Scientific Computing* 37, 3 (2015), C307–C330. https://doi.org/10.1137/14M0973773 arXiv:https://doi.org/10.1137/14M0973773

[29] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. (2018). http://arxiv.org/abs/1804.06826 arXiv: 1804.06826.

[30] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. 2017. *Designing SLATE: Software for Linear Algebra Targeting Exascale*. SLATE Working Notes 3, ICL-UT-17-06. Innovative Computing Laboratory, University of Tennessee.

[31] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA tensor core programmability, performance & precision. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018* (2018), 522–531. https://doi.org/10.1109/IPDPSW.2018.00091 arXiv: 1803.04014 ISBN: 9781538655559.

[32] Nvidia. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Technical Report. 53 pages. Issue: v1.1.

[33] Hiroyuki Ootomo. and Rio Yokota. 2019. TSQR on Tensor Cores. *SC '19, 29 The International Conference for High Performance Computing, Networking, Storage, and Analysis* (2019). https://doi.org/10.1145/1122445.1122456

[34] Christopher C. Paige and Michael A. Saunders. 1982. LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares. *ACM Trans. Math. Softw.* 8, 1 (March 1982), 43–71. https://doi.org/10.1145/355984.355989

[35] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols a. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Software* 39, 2 (2013), 1–24. https://doi.org/10.1145/2427023.2427030 arXiv: 1502.07526v1 ISBN: 9781577357384.

[36] R. Schreiber and C. Van Loan. 1989. A Storage-Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Statist. Comput.* 10, 1 (1989), 53–57. https://doi.org/10.1137/0910005

[37] Gilbert W Stewart. 1998. *Matrix Algorithms: Volume 1: Basic Decompositions*. Vol. 1. Siam.

[38] G. W. Stewart. 2008. Block Gram–Schmidt Orthogonalization. *SIAM Journal on Scientific Computing* 31, 1 (Jan. 2008), 761–775. https://doi.org/10.1137/070682563

[39] L. N. Trefethen and D. III Bau. 1997. *Numerical Linear Algebra*. Vol. 12. SIAM. https://doi.org/10.1137/1.9780898719574 Publication Title: Numerical Linear Algebra with Applications ISSN: 1070-5325.

[40] Å.Björck. 1994. Numerics ofGram-Schmidt orthogonalization. *Linear Algebra Appl.* 197-198, 1 (1994), 297–316. https://doi.org/10.1016/0024-3795(94)90493-6