ROUNDING ERROR ANALYSIS OF MIXED-PRECISION HOUSEHOLDER QR ALGORITHMS

L. MINAH YANG, ALYSON FOX, AND GEOFFREY SANDERS

Abstract. Although mixed precision arithmetic has recently garnered interest for training dense neural networks, many other applications could benefit from the speed-ups and lower storage if applied appropriately. The growing interest in employing mixed precision computations motivates the need for rounding error analysis that properly handles behavior from mixed precision arithmetic. We present a framework for mixed precision analysis that builds on the foundations of rounding error analysis presented in [12] and demonstrate its practicality by applying the analysis to various Householder QR Algorithms.

1. Introduction.

The accuracy of a numerical algorithm depends on several factors, including numerical stability and well-conditionedness of the problem, both of which may be sensitive to rounding errors, the difference between exact and finite-precision arithmetic. Low precision floats use fewer bits than high precision floats to represent the real numbers and naturally incur larger rounding errors. Therefore, error attributed to round-off may have a larger influence over the total error when using low precision, and some standard algorithms may yield insufficient accuracy when using low precision storage and arithmetic. However, many applications exist that would benefit from the use of lower precision arithmetic and storage that are less sensitive to floating-point round off error, such as clustering or ranking graph algorithms [?] or training dense neural networks [17], to name a few.

Many computing applications today require solutions quickly and often under low size, weight, and power constraints (low SWaP), e.g., sensor formation, etc. Computing in low-precision arithmetic offers the ability to solve many problems with improvement in all four parameters. Utilizing mixed-precision, one can achieve similar quality of computation as high-precision and still achieve speed, size, weight, and power constraint improvements. There have been several recent demonstrations of computing using half-precision arithmetic (16 bits) achieving around half an order to an order of magnitude improvement of these categories in comparison to double precision (64 bits). Trivially, the size and weight of memory required for a specific problem is 4×. Additionally, there exist demonstrations that the power consumption improvement is similar [?]. Modern accelerators (e.g., GPUs, Knights Landing, or Xeon Phi) are able to achieve this factor or better speedup improvements. Several examples include: (i) 2-4× speedup in solving dense large linear equations [10, 11], (ii) 12× speedup in training dense neural networks, and (iii) 1.2-10× speedup in small batched dense matrix multiplication [1] (up to 26× for batches of tiny matrices). Training deep artificial neural networks by employing lower precision arithmetic to various tasks such as multiplication [5] and storage [6] can easily be implemented on GPUs and are already a common practice in data science applications.

The low precision computing environments that we consider are *mixed precision* settings, which are designed to imitate those of new GPUs that employ multiple precision types for certain tasks. For example, Tesla V100's Tensor Cores perform matrix-multiply-and-accumulate of half precision input data with exact products and single precision (32 bits) summation accumulate [2]. The existing rounding error analyses are built within what we call a *uniform precision* setting, which is the assumption that all arithmetic operations and storage are performed via the same precision. In

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 17-SI-004, LLNL-JRNL-795525-DRAFT.

this work, we develop a framework for deterministic mixed-precision rounding error analysis, and explore half-precision Householder QR factorization (HQR) algorithms for data and graph analysis applications. QR factorization is known to provide a backward stable solution to the linear least squares problem and thus, is ideal for mixed-precision.

However, additional analysis is needed as the additional round-off error will effect orthogonality, and thus the accuracy of the solution. Here, we focus on analyzing specific algorithms in a specific set of types (IEEE754 half (fp16), single (fp32, and double(fp64)), but the framework we develop could be used on different algorithms or different floating point types (such as bfloat16 in [20]).

This work discusses several aspects of using mixed-precision arithmetic: (i) error analysis that can more accurately describe mixed-precision arithmetic than existing analyses, (ii) algorithmic design that is more resistant against lower numerical stability associated with lower precision types, and (iii) an example where mixed-precision implementation performs as sufficiently as double-precision implementations. Our key findings are that the new mixed-precision error analysis produces tighter error bounds, that some block QR algorithms by Demmel et al. [8] are able to operate in low precision more robustly than non-block techniques, and that some small-scale benchmark graph clustering problems can be successfully solved with mixed-precision arithmetic.

2. Background: Build up to rounding error analysis for inner products.

In this section, we introduce the basic motivations and tools for mixed-precision rounding error analysis needed for the *QR factorization.* A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ for $m \geq n$ can be written as

$$\mathbf{A} = \mathbf{Q}\mathbf{R}, \qquad \mathbf{Q} \in \mathbb{R}^{m\times m}, \qquad \mathbf{R} \in \mathbb{R}^{m\times n},$$

where $\mathbf{Q}$ is orthogonal, $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}_{m\times m}$, and $\mathbf{R}$ is upper trapezoidal. The above formulation is a *full* QR factorization, whereas a more efficient *thin* QR factorization results in $\mathbf{Q}_1 \in \mathbb{R}^{m\times n}$ and $\mathbf{R}_1 \in \mathbb{R}^{n\times n}$, that is

$$\mathbf{A} = \mathbf{Q}\mathbf{R} = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}_{m-n\times n}\end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1.$$

If $\mathbf{A}$ is full rank then the columns of $\mathbf{Q}_1$ are orthonormal (i.e. $\mathbf{Q}_1^\top\mathbf{Q}_1 = \mathbf{I}_{n\times n}$) and $\mathbf{R}_1$ is upper triangular. In many applications, computing the *thin* decomposition requires less computation and is sufficient in performance. While important definitions are stated explicitly in the text, Table 1 serves to establish basic notation.

Symbol(s)	Definition(s)	Section(s)
$\mathbf{x}$, $\mathbf{A}$	Vector, matrix	2
$\mathbf{Q}$	Orthogonal factor $\mathbf{A} \in \mathbb{R}^{m\times n}$: m-by-m (full) or m-by-n (thin)	2
$\mathbf{R}$	Upper triangular or trapezoidal factor of $\mathbf{A} \in \mathbb{R}^{m\times n}$: m-by-n (full) or n-by-n (thin)	2
$\mathrm{fl}(\mathbf{x})$, $\hat{\mathbf{x}}$	Quantity $\mathbf{x}$ calculated from floating point operations	2.1
b, t, μ, η	Base/precision/mantissa/exponent bits	2.1
k	Number of successive FLOPs	2.1
u^q	Unit round-off for precision t_q and base b_q: $\frac{1}{2}b_q^{1-t_q}$	2.1
δ^q	Quantity bounded by: $\lvert\delta^q\rvert < u^q$	2.1
γ_k^q, θ_k^q	$\frac{ku^q}{1-ku^q}$, Quantity bounded by: $\lvert\theta_k^q\rvert \leq \gamma_k^q$	2.1

TABLE 1
Basic definitions

Subsection 2.1 introduces basic concepts for rounding error analysis, and Subsection 2.2 exemplifies the need for mixed-precision rounding error analysis using the inner product.

2.1. Basic rounding error analysis of floating point operations. We use and analyze the IEEE 754 Standard floating point number systems. Let $\mathbb{F} \subset \mathbb{R}$ denote the space of some floating point number system with base $b \in \mathbb{N}$, precision $t \in \mathbb{N}$, significand $\mu \in \mathbb{N}$, and exponent range $[\eta_{\min}, \eta_{\max}] \subset \mathbb{Z}$. Then every element y in $\mathbb{F}$ can be written as

$$y = \pm\mu \times b^{\eta - t}, \tag{2.1}$$

where μ is any integer in $[0, b^t - 1]$ and η is an integer in $[\eta_{\min}, \eta_{\max}]$. While base, precision, and exponent range are fixed and define a floating point number,the sign, significand, and exponent identifies a unique number within that system. Although operations we use on $\mathbb{R}$ cannot be replicated exactly due to the finite cardinality of $\mathbb{F}$, we can still approximate the accuracy of analogous floating point operations (FLOPs). We adopt the rounding error analysis tools described in [12], which allow a relatively simple framework for formulating error bounds for complex linear algebra operations. A short analysis of FLOPs (see Theorem 2.2 [12]) shows that the relative error is controlled by the unit round-off, $u := \frac{1}{2}b^{1-t}$.

Name	b	t	# of exponent bits	$\eta_{\min}$	$\eta_{\max}$	unit round-off u
fp16 (IEEE754 half)	2	11	5	-15	16	4.883e-04
fp32 (IEEE754 single)	2	24	8	-127	128	5.960e-08
fp64 (IEEE754 double)	2	53	11	-1023	1024	1.110e-16

TABLE 2
IEEE754 formats and their primary attributes.

Let 'op' be any basic operation from the set $\mathrm{OP} = \{+, -, \times, \div\}$ and let $x, y \in \mathbb{R}$. The true value $(x \text{ op } y)$ lies in $\mathbb{R}$, and it is rounded using some conversion to a floating point number, $\mathrm{fl}(x \text{ op } y)$, admitting a rounding error. The IEEE 754 Standard requires *correct rounding*, which rounds the exact solution $(x \text{ op } y)$ to the closest floating point number and, in case of a tie, to the floating point number that has a mantissa ending in an even number. *Correct rounding* gives us an assumption for the error model where a single basic floating point operation yields a relative error, δ, bounded in the following sense:

$$\mathrm{fl}(x \text{ op } y) = (1+\delta)(x \text{ op } y), \quad |\delta| \le u, \quad \text{op} \in \{+, -, \times, \div\}. \tag{2.2}$$

We use (2.2) as a building block in accumulating errors from successive FLOPs. For example, consider computing $x+y+z$, where $x, y, z \in \mathbb{R}$ with a machine that can only compute one operation at a time. Then, there is a rounding error in computing $\hat{s_1} := \mathrm{fl}(x+y) = (1+\delta)(x+y)$, and another rounding error in computing $\hat{s_2} := \mathrm{fl}(\hat{s_1} + z) = (1+\tilde{\delta})(\hat{s_1} + z)$, where $|\delta|, |\tilde{\delta}| < u$. Then,

$$\mathrm{fl}(x+y+z) = (1+\tilde{\delta})(1+\delta)(x+y) + (1+\tilde{\delta})z. \tag{2.3}$$

Multiple successive operations introduce multiple rounding error terms, and keeping track of all errors is challenging. Lemma 2.1 introduces a convenient and elegant bound that simplifies accumulation of rounding error.

Lemma 2.1 (Lemma 3.1 [12]). *Let $|\delta_i| < u$ and $\rho_i \in \{-1, +1\}$, for $i = 1, \cdots, k$ and $ku < 1$. Then,*

$$\prod_{i=1}^{k} (1+\delta_i)^{\rho_i} = 1 + \theta_k, \qquad \textit{where} \qquad |\theta_k| \leq \frac{ku}{1-ku} =: \gamma_k. \tag{2.4}$$

We also use

$$\tilde{\gamma}_k = \frac{cku}{1-cku},$$

where $c > 0$ is a small integer and further extend this to θ so that $|\tilde{\theta}_k| \leq \tilde{\gamma}_k$.

In other words, θ_k represents the accumulation of rounding errors from k successive operations, and it is bounded by γ_k. Allowing θ_k's to be any arbitrary value within the corresponding γ_k bounds further aids in keeping a clear, simple error analysis. Applying this lemma to our example of adding three numbers results in

$$\mathrm{fl}(x+y+z) = (1+\tilde{\delta})(1+\delta)(x+y) + (1+\tilde{\delta})z = (1+\theta_2)(x+y) + (1+\theta_1)z. \tag{2.5}$$

Since $|\theta_1| \leq \gamma_1 < \gamma_2$, we can further simplify (2.5) to

$$\mathrm{fl}(x+y+z) = (1+\tilde{\theta}_2)(x+y+z), \quad \text{where} \quad |\tilde{\theta}_2| \leq \gamma_2, \tag{2.6}$$

at the cost of a slightly larger upper bound. Typically, error bounds formed in the fashion of (2.6) are converted to relative errors in order to put the error magnitudes in perspective. The relative error bound for our example is

$$\frac{|(x+y+z) - \mathrm{fl}(x+y+z)|}{|x+y+z|} \leq \gamma_2$$

when we assume $x + y + z \neq 0$.

Although Lemma 2.1 requires $ku < 1$, we actually need $ku < \frac{1}{2}$ to maintain a meaningful relative error bound as this assumption implies $\gamma_k < 1$ and guarantees a relative error below 100%. Since higher precision floating points have smaller unit round-off values, they can tolerate more successive FLOPs than lower precision floating points before reaching $\gamma_m = 1$. Table 3 shows the maximum number of successive floating point operations that still guarantees a relative error below 100% for various floating point types. Thus, accumulated rounding errors in lower precision types

precision	$\tilde{k} = \arg\max_k(\gamma_k \leq 1)$
FP16	`512`
FP32	$\approx$ `4.194e06`
FP64	$\approx$ `2.252e15`

Table 3

Upper limits of meaningful relative error bounds in the $\gamma^{(k)}$ notation.

can lead to an instability with fewer operations in comparison to higher precision types and prompts us to evaluate whether existing algorithms can be naively adapted for mixed-precision arithmetic.

2.2. Rounding Error Example for the Inner Product. We now consider computing the inner product of two vectors to clearly illustrate how this situation restricts rounding error analysis in fp16. An error bound for an inner product of m-length vectors is

$$|\mathbf{x}^\top \mathbf{y} - \mathrm{fl}(\mathbf{x}^\top \mathbf{y})| \leq \gamma_m |\mathbf{x}|^\top |\mathbf{y}|, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^m \tag{2.7}$$

as shown in [12]. While this result does not guarantee a high relative accuracy when $|\mathbf{x}^\top \mathbf{y}| \ll |\mathbf{x}|^\top |\mathbf{y}|$, high relative accuracy is expected in some special cases. For example, let $\mathbf{x} = \mathbf{y}$. Then we have exactly $|\mathbf{x}^\top \mathbf{x}| = |\mathbf{x}|^\top |\mathbf{x}| = \|\mathbf{x}\|_2^2$, which leads to a forward error: $\left| \|\mathbf{x}\|_2^2 - \mathrm{fl}(\|\mathbf{x}\|_2^2) \right| \leq \gamma_m \|\mathbf{x}\|_2^2$. Since vectors of length m accumulate rounding errors that are bounded by γ_m, the dot products of vectors computed in fp16 already face a 100% relative error bound in the worst-case scenario ($\gamma_{512}^{\text{fp16}} = 1$).

We present a simple numerical experiment that shows that the standard deterministic error bound is too pessimistic and cannot be practically used to approximate rounding error for half-precision arithmetic. In this experiment, we generated 2 million random half-precision vectors of length 512 from two random distributions: the standard normal distribution, $N(0,1)$, and the uniform distribution over $(0,1)$. Half precision arithmetic was simulated by calling alg. 1, which was proven to be a faithful simulation in [14], for every FLOP (multiplication and addition for the dot product). The relative error in this experiment is formulated as the LHS in Equation 2.7 divided by $|\mathbf{x}|^\top |\mathbf{y}|$ and all operations outside of calculating $\mathrm{fl}(\mathbf{x}^\top \mathbf{y})$ are executed by casting up to fp64 and using fp64 arithmetic. Table 4 shows some statistics from computing the relative error for simulated half precision dot products of 512-length random vectors. We see that the inner products of vectors sampled from the standard normal distribution have backward relative errors that do not deviate much from the unit round-off ($\mathcal{O}(\texttt{1e-4})$), whereas the vectors sampled from the uniform distribution tend to accumulate larger errors on average ($\mathcal{O}(\texttt{1e-3})$). Even so, the theoretical upper error bound of 100% is far too pessimistic as the maximum relative error does not even meet 2% in this experiment. Recent work in developing probabilistic bounds on rounding errors of floating point operations (see [13, 16]) have shown that the inner product relative backward error for the conditions used for this experiment is bounded by `5.466e-2` with probability 0.99.

Algorithm 1: $\mathbf{z}^{\text{fp16}} = \texttt{simHalf}(f, \mathbf{x}^{\text{fp16}}, \mathbf{y}^{\text{fp16}})$. Simulate function $f \in \mathrm{OP} \cup \{\texttt{dot\_product}\}$ in half precision arithmetic given input variables $\mathbf{x}, \mathbf{y}$. Function `castup` converts half precision floats to single precision floats, and `castdown` converts single precision floats to half precision floats by rounding to the nearest half precision float.

Input: $\mathbf{x}^{\text{fp16}}, \mathbf{y}^{\text{fp16}} \in \mathbb{F}^m_{\text{fp16}}$, $f : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}^n$

Output: $\mathrm{fl}(f(\mathbf{x}^{\text{fp16}}, \mathbf{y}^{\text{fp16}})) \in \mathbb{F}^n_{\text{fp16}}$

$\mathbf{x}^{\text{fp32}}, \mathbf{y}^{\text{fp32}} \leftarrow \texttt{castup}([\mathbf{x}^{\text{fp16}}, \mathbf{y}^{\text{fp16}}])$

$\mathbf{z}^{\text{fp32}} \leftarrow \mathrm{fl}(f(\mathbf{x}^{\text{fp32}}, \mathbf{y}^{\text{fp32}}))$

$\mathbf{z}^{\text{fp16}} \leftarrow \texttt{castdown}(\mathbf{z}^{\text{fp32}})$

return $\mathbf{z}^{\text{fp16}}$

Most importantly, no rounding error bounds (deterministic or probabilistic) allow flexibility in the precision types used for different operations. This restriction is the biggest obstacle in gaining an understanding of rounding errors to expect from computations done on emerging hardware that support mixed-precision such as GPUs that employ mixed-precision arithmetic.

Random Distribution	Average	Standard deviation	Maximum
Standard normal	1.627e-04	1.640e-04	2.838e-03
Uniform $(0, 1)$	2.599e-03	1.854e-03	1.399e-02

TABLE 4

Statistics from dot product backward relative error in for 512-length vectors stored in half-precision and computed in simulated half-precision from 2 million realizations.

We start by introducing some additional rules from [12] that build on Lemma 2.1 in Lemma 2.2. These rules summarize how to accumulate errors represented by θ's and γ's in a *uniform precision* setting. These relations aid in writing clear and simpler error analyses. Regardless of the specific details of a mixed-precision setting, a rounding error analysis for mixed-precision arithmetic must support at least two different precision types. Thus, Lemma 2.3 allows low and high precision types and is a simple modification of Lemma 2.2. The rules for θ allows us to keep track of the two precision types separately and the rules we present for γ were chosen to be useful for casting down to the lower of the two precisions, a pertinent procedure in our mixed-precision analysis in the later sections.

LEMMA 2.2. *For any positive integer k, let θ_k denote a quantity bounded according to $|\theta_k| \leq \frac{ku}{1-ku} =: \gamma_k$. The following relations hold for positive integers i, j, and nonnegative integer k. Arithmetic operations between θ_k's:*

$$(1+\theta_k)(1+\theta_j) = (1+\theta_{k+j}) \quad \text{and} \quad \frac{1+\theta_k}{1+\theta_j} = \begin{cases} 1+\theta_{k+j}, & j \leq k \\ 1+\theta_{k+2j}, & j > k \end{cases} \tag{2.8}$$

Operations on γ's:

$$\gamma_k\gamma_j \leq \gamma_{\min(\mathrm{k,j})}, \quad \text{for } \max_{(\mathrm{j,k})}\mathrm{u} \leq \frac{1}{2},$$
$$n\gamma_k \leq \gamma_{nk}, \quad \text{for} \quad n \leq \frac{1}{uk},$$
$$\gamma_k + u \leq \gamma_{k+1},$$
$$\gamma_k + \gamma_j + \gamma_k\gamma_j \leq \gamma_{k+j}.$$

LEMMA 2.3. *For any nonnegative integer k and some precision q, let θ_k^q denote a quantity bounded according to $|\theta_k^q| \leq \frac{ku^q}{1-ku^q} =: \gamma_k^q$. The following relations hold for two precisions l (low) and h (high), positive integers, j_l,j_h, non-negative integers k_l, and k_h, and $c > 0$:*

$$(1+\theta^l_{k_l})(1+\theta^l_{j_l})(1+\theta^h_{k_h})(1+\theta^h_{j_h}) = (1+\theta^l_{k_l+j_l})(1+\theta^h_{k_h+j_h}), \tag{2.9}$$

$$\frac{(1+\theta^l_{k_l})(1+\theta^h_{k_h})}{(1+\theta^l_{j_l})(1+\theta^h_{j_h})} = \begin{cases} (1+\theta^h_{k_h+j_h})(1+\theta^l_{k_l+j_l}), & j_h \leq k_h, j_l \leq k_l, \\ (1+\theta^h_{k_h+2j_h})(1+\theta^l_{k_l+j_l}), & j_h \leq k_h, j_l > k_l, \\ (1+\theta^h_{k_h+j_h})(1+\theta^l_{k_l+2j_l}), & j_h > k_h, j_l \leq k_l, \\ (1+\theta^h_{k_h+2j_h})(1+\theta^l_{k_l+2j_l}), & j_h > k_h, j_l > k_l. \end{cases} \tag{2.10}$$

Without loss of generality, let $1 \gg u_l \gg u_h > 0$. Let d, a nonnegative integer, and $r \in [0, \lfloor \frac{u_l}{u_h} \rfloor]$ be numbers that satisfy $k_h u_h = d u_l + r u_h$. Alternatively, d can be defined by $d := \lfloor \frac{k_h u_h}{u_l} \rfloor$. Then,

$$\gamma^h_{k_h} \gamma^l_{k_l} \leq \gamma^l_{k_l}, \quad \text{for } k_l u^l \leq \frac{1}{2} \tag{2.11}$$

$$\gamma^h_{k_h} + u^l \leq \gamma^l_{d+2} \tag{2.12}$$

$$\gamma^l_{k_l} + u^h \leq \gamma^l_{k_l+1} \tag{2.13}$$

$$\gamma^l_{k_l} + \gamma^h_{k_h} + \gamma^l_{k_l}\gamma^h_{k_h} < \gamma^l_{k_l+d+1}. \tag{2.14}$$

We use these principles to establish a mixed-precision rounding error analysis for computing the dot product, which is crucial in many linear algebra routines such as the QR factorization. Let us define a mixed-precision setting that is similar to the TensorCore Fused Multiply-Add (FMA) block but works at the level of a dot product. While the FMA block in TensorCore is for matrix-matrix products (level-3 BLAS), we consider a vector inner product(level-2 BLAS) FMA as defined in Assumption 2.4.

Assumption 2.4. *Let l and h each denote low and high precision types with unit round-off values u^l and u^h, where $1 \gg u^l \gg u^h > 0$. Consider an FMA operation for inner products that take vectors stored in precision l, compute products in full precision, and sum the products in precision h. Finally, the result is then cast back down to precision l.*

The full precision multiplication in Assumption 2.4 is exact when the low precision type is fp16 and the high precision type of fp32 due to their precisions and exponent ranges. As a quick proof, consider $x^{\text{fp16}} = \pm\mu_x 2^{\eta_x - 11}, y^{\text{fp16}} = \pm\mu_y 2^{\eta_y - 11}$ where $\mu_x, \mu_y \in [0, 2^{11}-1]$ and $\eta_x, \eta_y \in [-15, 16]$, and note that the significand and exponent range for fp32 are $[0, 2^{24}-1]$ and $[-127, 128]$. Then the product in full precision is

$$x^{\text{fp16}} y^{\text{fp16}} = \pm\mu_x\mu_y 2^{\eta_x+\eta_y+2-24},$$

where $\mu_x\mu_y \in [0, (2^{11}-1)^2] \subseteq [0, 2^{24}-1]$ and $\eta_x + \eta_y + 2 \in [-28, 34] \subseteq [-127, 128]$, and therefore is exact. Thus, the summation and the final cast down operations are the only sources of rounding error.

Let $\mathbf{x}^{\text{fp16}}, \mathbf{y}^{\text{fp16}}$ be m-length vectors stored in fp16, s_k b the k^{th} partial sum, and $\hat{s_k}$ be s_k computed with FLOPs. Then,

$$\begin{aligned}
\hat{s_1} &= \text{fl}(\mathbf{x}_1\mathbf{y}_1) = \mathbf{x}_1\mathbf{y}_1, \\
\hat{s_2} &= \text{fl}(\hat{s_1} + \mathbf{x}_2\mathbf{y}_2) = (\mathbf{x}_1\mathbf{y}_1 + \mathbf{x}_2\mathbf{y}_2)\,(1+\delta^h_1), \\
\hat{s_3} &= \text{fl}(\hat{s_2} + \mathbf{x}_3\mathbf{y}_3) = \left[(\mathbf{x}_1\mathbf{y}_1 + \mathbf{x}_2\mathbf{y}_2)\,(1+\delta^h_1) + \mathbf{x}_3\mathbf{y}_3\right](1+\delta^h_2).
\end{aligned}$$

We can see a pattern emerging. The error for a general m-length vector dot product is then

$$\hat{s_m} = (\mathbf{x}_1\mathbf{y}_1 + \mathbf{x}_2\mathbf{y}_2)\prod_{k=1}^{m-1}(1+\delta^h_k) + \sum_{i=3}^{n}\mathbf{x}_i\mathbf{y}_i\left(\prod_{k=i-1}^{m-1}(1+\delta^h_k)\right). \tag{2.15}$$

Using Lemma 2.1, we further simplify and form componentwise backward errors with

$$\text{fl}(\mathbf{x}^\top\mathbf{y}) = (\mathbf{x}+\Delta\mathbf{x})^\top\mathbf{y} = \mathbf{x}^\top(\mathbf{y}+\Delta\mathbf{y}), \quad \text{for } |\Delta\mathbf{x}| \leq \gamma^h_{m-1}|\mathbf{x}|,\ \ |\Delta\mathbf{y}| \leq \gamma^h_{m-1}|\mathbf{y}|. \tag{2.16}$$

Casting this down to fp16, then we incur a rounding error quantified by $d := \lfloor \frac{(m-1)u^h}{u^l} \rfloor$. The resulting componentwise backward errors are

$$\text{fl}(\mathbf{x}^\top \mathbf{y}) = (\mathbf{x} + \Delta\mathbf{x})^\top \mathbf{y} = \mathbf{x}^\top(\mathbf{y} + \Delta\mathbf{y}), \quad \text{for } |\Delta\mathbf{x}| \leq \gamma^l_{d+1}|\mathbf{x}|,\ |\Delta\mathbf{y}| \leq \gamma^l_{d+1}|\mathbf{y}|. \tag{2.17}$$

Equations (2.16) and (2.17) are crucial for our analysis in section 4 since the TensorCore technology outputs a matrix product in fp16 or fp32. Consider matrices $\mathbf{A} \in \mathbb{F}^{p\times m}_{\text{fp16}}$ and $\mathbf{B} \in \mathbb{F}^{m\times q}_{\text{fp16}}$, and $\mathbf{C} = \mathbf{AB} \in \mathbb{F}^{p\times q}_{\text{fp16}}$. If $\text{fl}(\mathbf{C})$ is desired in fp16, then each component of that matrix incurs rounding errors as shown in (2.17) and if it is desired in fp32, the componentwise rounding error is given by (2.16). Similarly, we could consider other mixed-precision algorithms that cast down at various points within the algorithm to take advantage of better storage properties of lower precision types. Error bounds in the fashion of (2.16) can be used before the cast down operations, and the action of the cast down is best represented by error bounds similar to (2.17).

In section 3, we introduce various Householder QR algorithms as well as a skeleton for rounding error analysis for these algorithms that we will modify for different mixed precision assumptions in section 4.

3. Algorithms and existing round-off error analyses.

We introduce the Householder QR factorization algorithm (HQR) in subsection 3.1 and two block variants that use HQR within the block in subsections 3.2 and 3.3. The blocked HQR (BQR) in subsection 3.2 partitions the columns of the target matrix and utilizes mainly level-3 BLAS operations and is a well-known algorithm that uses the WY representation of [4]. In contrast, the Tall-and-Skinny QR (TSQR) in subsection 3.3 partitions rows of the matrix and takes a communication-avoiding divide-and-conquer approach that can be easily parallelized (see [7]). We also present the crucial results in standard rounding error analysis of these algorithms that excludes any mixed-precision assumptions. These building steps of round-off error analysis will be easily tweaked for various mixed-precision assumptions in section 4.

3.1. Householder QR (HQR).

The HQR algorithm uses Householder transformations to zero out elements below the diagonal of a matrix (see [15]). We present this as zeroing out all but the first element of some vector, $\mathbf{x} \in \mathbb{R}^m$.

LEMMA 3.1. *Given vector $\mathbf{x} \in \mathbb{R}^m$, there exist Householder vector, $\mathbf{v}$, and —Householder transformation matrix, $\mathbf{P_v}$, such that $\mathbf{P_v}$ zeros out $\mathbf{x}$ below the first element.*

$$\begin{aligned} \sigma &= -\operatorname{sign}(\mathbf{x}_1)\|\mathbf{x}\|_2, \quad \mathbf{v} = \mathbf{x} - \sigma\hat{e_1}, \\ \beta &= \frac{2}{\mathbf{v}^\top\mathbf{v}} = -\frac{1}{\sigma\mathbf{v}_1}, \qquad \mathbf{P_v} = \mathbf{I}_m - \beta\mathbf{v}\mathbf{v}^\top. \end{aligned} \tag{3.1}$$

The transformed vector, $\mathbf{P_v x}$, has the same 2-norm as $\mathbf{x}$ since Householder transformations are orthogonal: $\mathbf{P_v x} = \sigma\hat{\mathbf{e}}_1$. In addition, $\mathbf{P_v}$ is symmetric and orthogonal, $\mathbf{P_v} = \mathbf{P_v}^\top = \mathbf{P_v}^{-1}$.

3.1.1. HQR: Algorithm.

Given $\mathbf{A} \in \mathbb{R}^{m\times n}$ and Lemma 3.1, HQR is done by repeating the following processes until only an upper triangle matrix remains. For $i = 1, 2, \cdots, n$,

Step 1) Compute $\mathbf{v}$ and β that zeros out the i^{th} column of $\mathbf{A}$ beneath a_{ii} (see alg. 2), and

Step 2) Apply $\mathbf{P_v}$ to the bottom right partition, $\mathbf{A}[i:m, i:n]$ (lines 4-6 of alg. 3).

Consider the following 4-by-3 matrix example adapted from [12]. Let $\mathbf{P_i}$ represent the i^{th}

Householder transformation of this algorithm.

$$\mathbf{A} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{\text{apply } \mathbf{P_1} \text{ to } \mathbf{A}} \left[\begin{array}{c|cc} \times & \times & \times \\ \hline 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{array}\right] \xrightarrow{\text{apply } \mathbf{P_2} \text{ to } \mathbf{P_1 A}}$$

$$\left[\begin{array}{cc|c} \times & \times & \times \\ 0 & \times & \times \\ \hline 0 & 0 & \times \\ 0 & 0 & \times \end{array}\right] \xrightarrow{\text{apply } \mathbf{P_3} \text{ to } \mathbf{P_2 P_1 A}} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix} = \mathbf{P_3 P_2 P_1 A} =: \mathbf{R}$$

Then, the $\mathbf{Q}$ factor for a full QR factorization is $\mathbf{Q} := \mathbf{P_1 P_2 P_3}$ since $\mathbf{P_i}$'s are symmetric, and the thin factors for a general matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ are

$$\mathbf{Q}_{\text{thin}} = \mathbf{P_1} \cdots \mathbf{P_n} \mathbf{I}_{m \times n} \quad \text{and} \quad \mathbf{R}_{\text{thin}} = \mathbf{I}_{m \times n}^{\top} \mathbf{P_n} \cdots \mathbf{P_1} \mathbf{A}. \tag{3.2}$$

Algorithm 2: β, $\mathbf{v}$, σ = $\texttt{hh\_vec}(\mathbf{x})$. Given a vector $\mathbf{x} \in \mathbb{R}^n$, return $\mathbf{v}$, β, σ that satisfy $(I - \beta \mathbf{v}\mathbf{v}^{\top})\mathbf{x} = \sigma \hat{e_1}$ and $\mathbf{v}_1 = 1$ (see [?, 12]).

Input: $\mathbf{x} \in \mathbb{R}^m$

Output: $\mathbf{v} \in \mathbb{R}^m$, and $\sigma, \beta \in \mathbb{R}$ such that $(I - \beta \mathbf{v}\mathbf{v}^{\top})\mathbf{x} = \pm \|\mathbf{x}\|_2 \hat{e_1} = \sigma \hat{e_1}$

```
1 v ← copy(x)
2 σ ← −sign(x_1)‖x‖_2
3 v_1 ← x_1 − σ
4 β ← −v_1/σ
5 return β, v/v_1, σ
```

Algorithm 3: $\mathbf{V}$, $\boldsymbol{\beta}$, $\mathbf{R}$ = $\texttt{HQR2}(A)$. A Level-2 BLAS implementation of the Householder QR algorithm. Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ where $m \geq n$, return matrix $\mathbf{V} \in \mathbb{R}^{m \times n}$, vector $\boldsymbol{\beta} \in \mathbb{R}^n$, and upper triangular matrix $\mathbf{R}$. An orthogonal matrix $\mathbf{Q}$ can be generated from $\mathbf{V}$ and $\boldsymbol{\beta}$, and $\mathbf{QR} = \mathbf{A}$.

Input: $A \in \mathbb{R}^{m \times n}$ where $m \geq n$.

Output: $\mathbf{V}$,$\boldsymbol{\beta}$, $\mathbf{R}$

```
1 V, β ← 0_{m×n}, 0_m
2 for i = 1 : n do
3 |  v, β, σ ← hh_vec(A[i : end, i])                        /* Algorithm 2 */
4 |  V[i : end, i], β_i, A[i, i] ← v, β, σ
5 |  A[i + 1 : end, i] ← zeros(m − i)
6 |_ A[i : end, i + 1 : end] ← A[i : end, i + 1 : end] − βvv^⊤A[i : end, i + 1 : end]
7 return V, β, A[1 : n, 1 : n]
```

3.1.2. HQR: Rounding Error Analysis. Now we present an error analysis for alg. 3 by keeping track of the different operations of alg. 2 and alg. 3.

Calculating the i^{th} Householder vector and constant. In alg. 3, the i^{th} Householder vector shares all but the first component with the target column, $\mathbf{A}[i:m,i]$. We first calculate σ as is implemented in line 2 of alg. 2.

$$\mathrm{fl}(\sigma) = \hat{\sigma} = \mathrm{fl}(-\mathrm{sign}(\mathbf{A}_{\mathrm{i,i}})\|\mathbf{A}[\mathrm{i:m,i}]\|_2) = \sigma + \Delta\sigma, \quad |\Delta\sigma| \leq \gamma_{\mathrm{m-i+1}}|\sigma|. \tag{3.3}$$

Note that the backward error incurred here is simply that an inner product of a vector in $\mathbb{R}^{m-i+1}$ with itself. Let $\tilde{\mathbf{v}}_1 \equiv \mathbf{A}_{i,i} - \sigma$, the penultimate value $\mathbf{v}_1$. The subtraction adds a single additional rounding error via

$$\mathrm{fl}(\tilde{\mathbf{v}}_1) = \tilde{\mathbf{v}}_1 + \Delta\tilde{\mathbf{v}}_1 = (1+\delta)(\mathbf{A}_{i,i} - \sigma - \Delta\sigma) = (1+\tilde{\theta}_{m-i+2})(\mathbf{A}_{i,i} - \sigma)$$

where the last equality is granted because the sign of σ is chosen to prevent cancellation. For the sake of simplicity, we write $|\Delta\tilde{\mathbf{v}}_1| \leq \tilde{\gamma}_{m-i+1}|\tilde{\mathbf{v}}_1|$ even though a tighter relative upper bound is θ_{m-i+2} We sweep that minor difference (in comparison to $\mathcal{O}(m-i)$) under the our use of the $\tilde{\gamma}$ notation defined in Lemma 2.1. Since alg. 2 normalizes the Householder vector so that its first component is 1, the remaining components of $\mathbf{v}$ are divided by $\mathrm{fl}(\tilde{\mathbf{v}}_1)$ incurring another single rounding error. As a result, the rounding errors in $\mathbf{v}$ are

$$\mathrm{fl}(\mathbf{v}_j) = \mathbf{v}_j + \Delta\mathbf{v}_j \text{ where } |\Delta\mathbf{v}_j| \leq \begin{cases} 0, & j = 1 \\ \tilde{\gamma}_{m-i+1}|\mathbf{v}_j|, & j = 2 : m-i+1. \end{cases} \tag{3.4}$$

Next, we consider the Householder constant, β, as is computed in line 4 of alg. 2.

$$\hat{\beta} = \mathrm{fl}\left(-\frac{\tilde{\mathbf{v}_1}}{\hat{\sigma}}\right) = -(1+\delta)\frac{\tilde{\mathbf{v}}_1 + \boldsymbol{\Delta}\tilde{\mathbf{v}}_1}{\sigma + \Delta\sigma} \tag{3.5}$$

$$= \frac{(1+\delta)(1+\theta_{m-i+1})}{(1+\theta_{m-i+2})}\beta = (1+\theta_{3(m-i+2)})\beta \tag{3.6}$$

$$= \beta + \Delta\beta, \text{ where } |\Delta\beta| \leq \tilde{\gamma}_{m-i+1}\beta. \tag{3.7}$$

We have shown (3.5) to keep our analysis simple in section 4 and (3.6) and (3.7) show that the error incurred from calculating of $\|\mathbf{A}[i:m,i]\|_2$ accounts for the vast majority of the rounding error so far.

Applying a Single Householder Transformation. Now we consider lines 4-6 of alg. 3. Since the entries in $\mathbf{A}[i+1:m,i]$ are simply zeroed out and $\mathbf{A}_{i,i}$ is replaced by σ, we only need to calculate the errors for applying a Householder transformation with the computed Householder vector and constant. This is the most crucial building block of the rounding error analysis for any variant of HQR because the $\mathbf{Q}$ factor is formed by applying the Householder transformations to the identity and both of the blocked versions in subsection 3.2 and subsection 3.3 require efficient implementations of this step. In this section, we only consider a level-2 BLAS implementation of applying the Householder transformation, but in subsection 3.2 we introduce a level-3 BLAS implementation.

A Householder transformation is applied through a series of inner and outer products, since Householder matrices are rank-1 updates of the identity. That is, computing $\mathbf{P}_{\mathbf{v}}\mathbf{x}$ for any $\mathbf{x} \in \mathbb{R}^m$ is as simple as computing

$$\mathbf{y} := \mathbf{x} - (\beta\mathbf{v}^\top\mathbf{x})\mathbf{v}. \tag{3.8}$$

Let us assume that $\mathbf{x}$ is an exact vector and there were errors incurred in forming $\mathbf{v}$ and β. The errors incurred from computing $\mathbf{v}$ and β need to be included in addition to the new rounding errors accumulating from the action of applying $\mathbf{P_v}$ to a column. In practice, $\mathbf{x}$ would be a column in $\mathbf{A}^{(i-1)}[i+1:m, i+1:n]$, where the superscript $(i-1)$ indicates that this submatrix of $\mathbf{A}$ has already been transformed by $i-1$ Householder transformations that zeroed out components below $\mathbf{A}_{j,j}$ for $j = 1 : i-1$. We show the error for forming $\mathrm{fl}\left(\hat{\mathbf{v}}^\top\mathbf{x}\right)$ where we continue to let $\mathbf{v}, \mathbf{x} \in \mathbb{R}^{m-i+1}$ as would be in the i^{th} iteration of the for-loop in alg. 3:

$$\mathrm{fl}\left(\hat{\mathbf{v}}^\top\mathbf{x}\right) = (1+\theta_{m-i+1})(\mathbf{v}+\mathbf{\Delta v})^\top\mathbf{x}.$$

Set $\mathbf{w} := \beta\mathbf{v}^\top\mathbf{x}\mathbf{v}$. Then,

$$\hat{\mathbf{w}} = (1+\theta_{m-i+1})(1+\delta)(1+\tilde{\delta})(\beta+\Delta\beta)(\mathbf{v}+\mathbf{\Delta v})^\top\mathbf{x}(\mathbf{v}+\mathbf{\Delta v}),$$

where θ_{m-i+1} is from computing the inner product $\hat{\mathbf{v}}^\top\mathbf{x}$, and δ and $\tilde{\delta}$ are from multiplying β, $\mathrm{fl}(\hat{\mathbf{v}}^\top\mathbf{x})$, and $\hat{\mathbf{v}}$ together. Finally, we can add in the vector subtraction operation and complete the rounding error analysis of applying a Householder transformation to any vector:

$$\mathrm{fl}(\hat{\mathbf{P_v}}\mathbf{x}) = \mathrm{fl}(\mathbf{x}-\hat{\mathbf{w}}) = (1+\delta)(\mathbf{x}-\mathbf{w}-\mathbf{\Delta w}) = \mathbf{y}+\mathbf{\Delta y}, \tag{3.9}$$

where $|\mathbf{\Delta y}| \leq u|\mathbf{x}| + \tilde{\gamma}_{m-i+1}|\beta||\mathbf{v}||\mathbf{v}|^\top|\mathbf{x}|$. Using $\sqrt{2/\beta} = \|\mathbf{v}\|_2$, we can conclude

$$\|\mathbf{\Delta y}\|_2 \leq \tilde{\gamma}_{m-i+1}\|\mathbf{x}\|_2. \tag{3.10}$$

Next, we convert this to a backward error for $\mathbf{P_v}$. Since $\mathbf{\Delta P_v}$ is exactly $\frac{1}{\mathbf{x}^\top\mathbf{x}}\mathbf{\Delta y}\mathbf{x}^\top$, we can compute its Frobenius norm by using $\mathbf{\Delta P}_{ij} = \frac{1}{\|\mathbf{x}\|_2^2}\mathbf{\Delta y}_i\mathbf{x}_j$,

$$\|\mathbf{\Delta P}\|_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{m}\left(\frac{1}{\|\mathbf{x}\|_2^2}\mathbf{\Delta y}_i\mathbf{x}_j\right)^2\right)^{1/2} = \frac{\|\mathbf{\Delta y}\|_2}{\|\mathbf{x}\|_2} \leq \tilde{\gamma}_{m-i+1},$$

where the last inequality is a direct application of (3.10). We summarize these results in Lemma 3.2.

Lemma 3.2. *Let $\mathbf{x} \in \mathbb{R}^m$ and consider the computation of $\hat{\mathbf{y}} = \mathrm{fl}(\mathbf{P_v}\mathbf{x})$ via*

$$\mathbf{y}+\mathbf{\Delta y} = (\mathbf{P_v}+\mathbf{\Delta P_v})\mathbf{x} = \mathrm{fl}(\mathbf{P_v}\mathbf{x}) = \mathrm{fl}(\mathbf{x}-\hat{\beta}\hat{\mathbf{v}}\hat{\mathbf{v}}^\top\mathbf{x})$$

and rounding errors incurred in forming $\hat{\mathbf{v}}$ and $\hat{\beta}$ are expressed componentwise via $\hat{\mathbf{v}} = \mathbf{v}+\Delta\mathbf{v}$ and $\hat{\beta} = \beta+\Delta\beta$ where the relative componentwise errors for both are bounded by quantity γ_y:

$$|\Delta\mathbf{v}| \leq \gamma_y|\mathbf{v}|, \qquad |\Delta\beta| \leq \gamma_y|\beta|. \tag{3.11}$$

Then, the normwise forward and backward errors are $\|\Delta\mathbf{y}\|_2 \leq \gamma_y\|\mathbf{y}\|_2$ and $\|\mathbf{\Delta P_v}\|_F \leq \gamma_y$.

Note that in a uniform precision setting this bound is represented as $\gamma_y = \tilde{\gamma}_m$.

Applying many successive Householder transformations. Consider applying a sequence of transformations in the set $\{\mathbf{P_i}\}_{i=1}^r \subset \mathbb{R}^{m\times m}$ to $\mathbf{x} \in \mathbb{R}^m$, where $\mathbf{P_i}$'s are all Householder transformations computed with $\tilde{\mathbf{v}}_\mathbf{i}$'s and $\hat{\beta}_i$'s. This is directly applicable to HQR as $\mathbf{Q} = \mathbf{P_1}\cdots\mathbf{P_n}\mathbf{I}$ and $\mathbf{R} = \mathbf{Q}^\top\mathbf{A} = \mathbf{P_n}\cdots\mathbf{P_1}\mathbf{A}$. Lemma 3.3 is very useful for any sequence of transformations, where each transformation has a known bound. We will invoke this lemma to prove Lemma 3.4, and use it in future sections for other sequential transformations.

LEMMA 3.3. *If* $\mathbf{X_j} + \mathbf{\Delta X_j} \in \mathbb{R}^{m\times m}$ *satisfies* $\|\mathbf{\Delta X_j}\|_F \leq \delta_j \|\mathbf{X_j}\|_2$ *for all* j*, then*

$$\left\|\prod_{j=1}^{n}(\mathbf{X_j} + \mathbf{\Delta X_j}) - \prod_{j=1}^{n}\mathbf{X_j}\right\|_F \leq \left(-1 + \prod_{j=1}^{n}(1+\delta_j)\right)\prod_{j=1}^{n}\|\mathbf{X_j}\|_2.$$

LEMMA 3.4. *Consider applying a sequence of transformations* $\mathbf{Q} = \mathbf{P_r}\cdots\mathbf{P_2}\mathbf{P_1}$ *onto vector* $\mathbf{x} \in \mathbb{R}^m$ *to form* $\hat{\mathbf{y}} = \mathrm{fl}(\hat{\mathbf{P_r}}\cdots\hat{\mathbf{P_2}}\hat{\mathbf{P_1}}\mathbf{x})$*, where* $\hat{\mathbf{P_k}}$*'s are Householder transformations constructed from* $\hat{\beta}_k$ *and* $\hat{\mathbf{v_k}}$*. These Householder vectors and constants are computed via alg. 2 and the rounding errors are bounded by (3.4) and (3.7). If each transformation is computed via (3.8), then*

$$\hat{\mathbf{y}} = \mathbf{Q}(\mathbf{x} + \mathbf{\Delta x}) = (\mathbf{Q} + \mathbf{\Delta Q})\mathbf{x}, \tag{3.12}$$

$$\|\mathbf{\Delta y}\|_2 \leq r\tilde{\gamma}_m\|\mathbf{x}\|_2, \;\; \|\mathbf{\Delta Q}\|_F \leq r\tilde{\gamma}_m. \tag{3.13}$$

Proof. Applying Lemma 3.3 directly to $\mathbf{Q}$ yields

$$\|\hat{\mathbf{Q}} - \mathbf{Q}\|_F = \left\|\prod_{j=1}^{r}(\mathbf{P_j} + \mathbf{\Delta P_j}) - \prod_{j=1}^{r}\mathbf{P_j}\right\|_F \leq (1+\tilde{\gamma}_m)^r - 1\prod_{j=1}^{n}\|\mathbf{P_j}\|_2 \leq (1+\tilde{\gamma}_m)^r - 1$$

since $\mathbf{P_j}$'s are orthogonal and have 2-norm, 1. While we omit the details here, we can show that $(1+\tilde{\gamma}_m)^r - 1 \leq r\tilde{\gamma}_m$ using the argument for Lemma 2.1 if $r\tilde{\gamma}_m \leq 1/2$. ∎

In this current uniform precision error analysis, the important quantity $\tilde{\gamma}_m$ is derived from the backward error of applying one Householder transformation. To easily generalize this section for mixed-precision analysis, we benefit from alternatively denoting this quantity as $\tilde{\gamma}_{\mathbf{P}}$ with the understanding that $\tilde{\gamma}_{\mathbf{P}}$ will be some combination of $\tilde{\gamma}$'s of differing precisions. The bound in Equation (3.13) would then be replaced by $r\tilde{\gamma}_{\mathbf{P}}$. Applying Lemma 3.2 directly to columns of $\mathbf{A}$ and $\mathbf{I}$ allows us to formulate 2-norm forward bounds for columns of $\mathbf{R}$ and $\mathbf{Q}$. We show how to convert these columnwise bounds into matrix norms for the $\mathbf{R}$ factor.

$$\|\mathbf{\Delta R}\|_F = \left(\sum_{i=1}^{n}\|\mathbf{\Delta R}[:,i]\|_2^2\right)^{1/2} \leq \left(\sum_{i=1}^{n} n^2\tilde{\gamma}_m^2\|\mathbf{A}[:,i]\|_2^2\right)^{1/2} = n\tilde{\gamma}_m\|\mathbf{A}\|_F,$$

We gather these results into Theorem 3.5.

THEOREM 3.5. *Let* $\mathbf{A} \in \mathbb{R}^{m\times n}$ *with* $m \geq n$ *have full rank,* n*. Let* $\hat{\mathbf{Q}} \in \mathbb{R}^{m\times n}$ *and* $\hat{\mathbf{R}} \in \mathbb{R}^{n\times n}$ *be the thin QR factors of* $\mathbf{A}$ *obtained via alg. 3. Then,*

$$\hat{\mathbf{R}} = \mathbf{R} + \mathbf{\Delta R} = \mathrm{fl}(\hat{\mathbf{P}}_n\cdots\hat{\mathbf{P}}_1\mathbf{A}), \;\; \|\mathbf{\Delta R}[:,j]\|_2 \leq n\tilde{\gamma}_m\|\mathbf{A}[:,j]\|_2, \;\; \|\mathbf{\Delta R}\|_F \leq n\tilde{\gamma}_m\|\mathbf{A}\|_F$$

$$\hat{\mathbf{Q}} = \mathbf{Q} + \mathbf{\Delta Q} = \mathrm{fl}(\hat{\mathbf{P}}_1\cdots\hat{\mathbf{P}}_n\mathbf{I}), \;\; \|\mathbf{\Delta Q}[:,j]\|_2 \leq n\tilde{\gamma}_m, \;\; \|\mathbf{\Delta Q}\|_F \leq n^{3/2}\tilde{\gamma}_m.$$

Let $\mathbf{A} + \mathbf{\Delta A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$*, where* $\hat{\mathbf{Q}}$ *and* $\hat{\mathbf{R}}$ *are obtained via Algorithm 3. Then the backward error is*

$$\|\mathbf{\Delta A}\|_F \leq n^{3/2}\tilde{\gamma}_m\|\mathbf{A}\|_F. \tag{3.14}$$

The content of this section generalizes the standard rounding error analysis in [12] by employing quantities denoted via $\Delta\beta$, $\mathbf{\Delta v}$, $\tilde{\gamma}_y$, and $\tilde{\gamma}_{\mathbf{P}}$. These quantities account for various forward and backward errors formed in computing essential components of HQR, namely the Householder constant and vector, as well as normwise errors of the action of applying Householder transformations. In the next sections, we present blocked variants of HQR that use alg. 3.

3.2. Block HQR with partitioned columns (BQR).

We refer to the blocked variant of HQR where the columns are partitioned as BQR. Note that this algorithm relies on the WY representation described in [4] instead of the storage-efficient version of [19].

3.2.1. The WY Representation.

A convenient matrix representation that accumulates r Householder reflectors is known as the WY representation (see [4, 9]). Lemma 3.6 shows how to update a rank-j update of the identity, $\mathbf{Q}^{(j)}$, with a Householder transformation, $\mathbf{P}$, to produce a rank-$(j+1)$ update of the identity, $\mathbf{Q}^{(j+1)}$. With the correct initialization of $\mathbf{W}$ and $\mathbf{Y}$, we can build the WY representation of successive Householder transformations as shown in Algorithm 4. This algorithm assumes that the Householder vectors, $\mathbf{V}$, and constants,$\boldsymbol{\beta}$, have already been computed. Since the $\mathbf{Y}$ factor is exactly $\mathbf{V}$, we only need to compute the $\mathbf{W}$ factor.

LEMMA 3.6. *Suppose* $\mathbf{Q}^{(j)} = \mathbf{I} - \mathbf{W}^{(j)}\mathbf{Y}^{(j)\top} \in \mathbb{R}^{m\times m}$ *is an orthogonal matrix with* $\mathbf{W}^{(j)}, \mathbf{Y}^{(j)} \in \mathbb{R}^{m\times j}$*. Let us define* $\mathbf{P} = \mathbf{I} - \beta\mathbf{v}\mathbf{v}^\top$ *for some* $\mathbf{v} \in \mathbb{R}^m$ *and let* $\mathbf{z}^{(j+1)} = \beta\mathbf{Q}^{(j)}\mathbf{v}$*. Then,*

$$\mathbf{Q}^{(j+1)} = \mathbf{Q}^{(j)}\mathbf{P} = \mathbf{I} - \mathbf{W}^{(j+1)}\mathbf{Y}^{(j+1)\top},$$

where $\mathbf{W}^{(j+1)} = [\mathbf{W}^{(j)}|\mathbf{z}]$ *and* $\mathbf{Y}^{(j+1)} = [\mathbf{Y}^{(j)}|\mathbf{v}]$ *are each* m*-by-*$(j+1)$*.*

Algorithm 4: $\mathbf{W}, \mathbf{Y} \leftarrow$ `buildWY`$(V, \boldsymbol{\beta})$: Given a set of householder vectors $\{\mathbf{V}[:, i]\}_{i=1}^r$ and their corresponding constants $\{\boldsymbol{\beta}_i\}_{i=1}^r$, form the final $\mathbf{W}$ and $\mathbf{Y}$ factors of the WY representation of $\mathbf{P}_1 \cdots \mathbf{P}_r$, where $\mathbf{P}_i := \mathbf{I}_m - \boldsymbol{\beta}_i\mathbf{v}_i\mathbf{v}_i^\top$

Input: $\mathbf{V} \in \mathbb{R}^{m\times r}$, $\boldsymbol{\beta} \in \mathbb{R}^r$ where $m > r$.
Output: $\mathbf{W}$

Initialize: $\mathbf{W} := \boldsymbol{\beta}_1\mathbf{V}[:, 1]$. /* $\mathbf{Y}$ is $\mathbf{V}$. */
for $j = 2 : r$ **do**
  $\mathbf{z} \leftarrow \boldsymbol{\beta}_j \left[\mathbf{V}[:, j] - \mathbf{W}\left(\mathbf{V}[:, 1 : j-1]^\top\mathbf{V}[:, j]\right)\right]$
  $\mathbf{W} \leftarrow [\mathbf{W} \quad \mathbf{z}]$ /* Update $\mathbf{W}$ to an m-by-j matrix. */
return $\mathbf{W}$

In HQR, $\mathbf{A}$ is transformed into an upper triangular matrix $\mathbf{R}$ by identifying a Householder transformation that zeros out a column below the diagonal, then applying that Householder transformation to the bottom right partition. For example, the k^{th} Householder transformation finds an $m-k+1$ sized Householder transformation that zeros out column k below the diagonal and then applies it to the $(m-k+1)$-by-$(n-k)$ partition of the matrix, $\mathbf{A}[k : m, k+1 : n]$. Since the $k+1^{st}$ column is transformed by the k^{th} Householder transformation, this algorithm must be executed serially as shown in alg. 3. The highest computational burden at each iteration falls on alg. 3 line 6, which requires Level-2 BLAS operations when computed efficiently.

In contrast, BQR replaces this step with Level-3 BLAS operations by partitioning $\mathbf{A}$ by groups of columns. Let $\mathbf{A} = [\mathbf{C}_1 \cdots \mathbf{C}_N]$ where $\mathbf{C}_1, \cdots, \mathbf{C}_{N-1}$ are each m-by-r, and $\mathbf{C}_N$ holds the remaining columns. The k^{th} block, $\mathbf{C}_k$, is transformed using HQR (alg. 3) while building the WY representation of $\mathbf{P}_{(k-1)r+1} \cdots \mathbf{P}_{kr} = \mathbf{I}_m - \mathbf{W_k}\mathbf{Y_k}^\top$ as in alg. 4. Note that both algs. 3 and 4 are rich in Level-2 BLAS operations. Then, $\mathbf{I} - \mathbf{Y_k}\mathbf{W_k}^\top$ is applied to $[\mathbf{C}_2 \cdots \mathbf{C}_N]$ with two Level-3 BLAS operations as shown in line 6 of alg. 5. BQR performs approximately $1 - \mathcal{O}(1/N)$ fraction of its FLOPs in Level-3 BLAS operations (see section 5.2.3 of [9]), and can reap the benefits from the accelerated matrix-matrix-multiply and accumulate technology of TensorCore. Note that BQR does require more FLOPs when compared to HQR, but these additional FLOPs are negligible in high precision. A pseudoalgorithm for BQR is shown in alg. 5. Note that the subscripts on $\mathbf{W_i}$ indicate the WY representation for the Householder transformations on the i^{th} block of $\mathbf{A}$, $\mathbf{C_k}$, whereas the superscripts on $\mathbf{W}^{(j)}$ in Lemma 3.6 refers to the j^{th} update within building a WY representation.

Algorithm 5: $\mathbf{Q}, \mathbf{R} \leftarrow$ `blockHQR`$(\mathbf{A}, r)$: Perform Householder QR factorization of matrix $\mathbf{A}$ with column partitions of size r.

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$, $r \in \mathbb{R}$ where $r < n$.
Output: $\mathbf{Q}, \mathbf{R}$

1 $N = \lceil \frac{n}{r} \rceil$
   `// Let` $\mathbf{A} = [\mathbf{C_1} \cdots \mathbf{C_N}]$ `where all blocks except` $\mathbf{C_N}$ `are` m`-by-`r `sized.`
2 **for** $i = 1 : N$ **do**
3   $\mathbf{V_i}, \boldsymbol{\beta}_i, \mathbf{C_i} \leftarrow$ `hhQR`$(\mathbf{C_i})$ `/* Algorithm 3 */`
4   $\mathbf{W_i} \leftarrow$ `buildWY`$(\mathbf{V_i}, \boldsymbol{\beta}_i)$ `/* Algorithm 4 */`
5   **if** $i < N$ **then**
6     $[\mathbf{C_{i+1}} \cdots \mathbf{C_N}]$ -= $\mathbf{V_i}\left(\mathbf{W_i}^\top [\mathbf{C_{i+1}} \cdots \mathbf{C_N}]\right)$ `/* update the rest: BLAS-3 */`
   `//` $\mathbf{A}$ `has been transformed into` $\mathbf{R} = \mathbf{Q}^\top \mathbf{A}$`.`
   `// Now build` $\mathbf{Q}$ `using level-3 BLAS operations.`
7 $\mathbf{Q} \leftarrow \mathbf{I}$ `/*` $\mathbf{I}_m$ `if full QR, and` $\mathbf{I}_{m\times n}$ `if thin QR. */`
8 **for** $i = N : -1 : 1$ **do**
9   $\mathbf{Q}[(i-1)r+1 : m, (i-1)r+1 : n]$-= $\mathbf{W}_i \left(\mathbf{V}_i^\top \mathbf{Q}[(i-1)r+1 : m, (i-1)r+1 : n]\right)$
10 **return** $\mathbf{Q}, \mathbf{A}$

3.2.2. BQR: Rounding Error Analysis. We now present the basic structure for the rounding error analysis for alg. 5, which consist of: 1)HQR, 2)building the W factor, and 3) updating the remaining blocks with the WY representation. We have adapted the analysis from [12] to fit this exact variant. Furthermore, we assume that n is divisible by r so that $N = \lceil n/r \rceil = n/r$ to make our error analysis simple. In practice, an efficient implementation might require r to be a power of two or a product of small prime factors and result a thinner N^{th} block compared to the rest. This discrepancy is easily fixed by padding the matrix with zeros, a standard procedure for standard algorithms like the Fast Fourier Transform (FT).

HQR within each block: line 3 of alg. 5. We apply Algorithm 3 to the k^{th} block, $\mathbf{C}_k$, which is equivalent to r Householder transformations of size $m-(r-1)k$ under the assumption that $n = Nr$. A normwise bound for the $\mathbf{R}$ factor of $\mathbf{C_k}$ can be easily adapted from Theorem 3.5.

Build WY at each block: line 4 of alg. 5. We now calculate the rounding errors incurred from building the WY representation when given a set of Householder vectors and constants as shown in alg. 4. Our goal is to analyze the error accumulated from updating the WY representation from the $j-1^{st}$ step to the j^{th} for block $\mathbf{C_k}$. Let us represent the j^{th} Householder constant and vector of the k^{th} block computed with FLOPs as with $\hat{\beta}_k^{(j)}$ and $\hat{\mathbf{v}}_\mathbf{k}{}^{(j)}$ and the j'^{th} update to the WY representation as

$$\mathbf{X_k}^{(j')} = \mathbf{I} - \hat{\mathbf{W}}_\mathbf{k}{}^{(j')} \hat{\mathbf{Y}}_\mathbf{k}{}^{(j')\top}.$$

This action applies a rank-1 update via the subtraction of the outer product $\hat{\mathbf{z_k}}^{(j)}\hat{\mathbf{v_k}}^{(j)\top}$ to apply $\hat{\mathbf{P}}_{\mathbf{(k-1)r+j}}$ on the right. Since $\mathbf{z_k}^{(j)} = \beta_k^j \mathbf{X_k}^{(j-1)}\mathbf{v_k}^{(j)}$, the update performs a single Householder transformation in the same efficient implementation that is discussed in Lemma 3.2, but on the right side:

$$\begin{aligned}
\mathbf{X_k}^{(j)} &= \mathbf{X_k}^{(j-1)} - \beta_k^j \mathbf{X_k}^{(j-1)}\mathbf{v_k}^{(j)}\mathbf{v_k}^{(j-1)\top} \\
&= \mathbf{X_k}^{(j-1)}(\mathbf{I} - \beta_k^{(j)}\mathbf{v_k}^{(j)}\mathbf{v_k}^{(j-1)\top}) = \mathbf{X_k}^{(j-1)}\mathbf{P}_{\mathbf{(k-1)r+j}}.
\end{aligned}$$

Therefore, we apply (3.13) directly for the construction of $\mathbf{z_k}^{(j)} = \mathbf{Q_k}^{(j-1)}\mathbf{v_k^{(j)}}$ and result in Lemma 3.7.

LEMMA 3.7. *Consider the construction of the WY representation for the k^{th} partition of matrix $\mathbf{A}^{m\times n}$ given a set of Householder constants and vectors, $\{\beta_k^{(j)}\}_{j=1}^r$ and $\{\mathbf{v_k}^{(j)}$ via alg. 4. Then,*

$$\begin{aligned}
\hat{\mathbf{z_k}}^{(j)} &= \mathbf{z_k}^{(j)} + \mathbf{\Delta z_k}^{(j)}, \quad |\mathbf{\Delta z_k}^{(j)}| \leq j\tilde{\gamma}_{m-(k-1)r}|\mathbf{z_k}^{(j)}| \\
\hat{\mathbf{v_k}}^{(j)} &= \mathbf{v_k}^{(j)} + \mathbf{\Delta v_k}^{(j)}, \quad |\mathbf{\Delta v_k}^{(j)}| \leq \tilde{\gamma}_{m-(k-1)r}|\mathbf{v_k}^{(j)}|,
\end{aligned}$$

where the second bound is derived from (3.4).

Most importantly, this shows that constructing the WY update is just as numerically stable as applying successive Householder transformations (see Section 19.5 of [12]).

Update blocks to the right: line 6 of alg. 5. We now consider applying $\mathbf{X_k} := \mathbf{I} - \mathbf{W_k}\mathbf{Y_k}^\top$ to the bottom right submatrix, $\mathbf{B} := [\mathbf{C_{k+1}} \cdots \mathbf{C_N}][(k-1)r+1 : m, :]$. In practice, this step is performed with a level-3 BLAS operation. Regardless, let us analyze the column-wise backward error for the j^{th} column of $\mathbf{B}$, $\mathbf{b_j}$.

$$\mathrm{fl}(\hat{\mathbf{X}}_\mathbf{k}\mathbf{b_j}) = \mathrm{fl}(\mathbf{b_j} - \mathrm{fl}(\hat{\mathbf{W}}_\mathbf{k}\mathrm{fl}(\hat{\mathbf{Y}}_\mathbf{k}{}^\top\mathbf{b_j}))) = (1+\delta)(\mathbf{b_j} - (\hat{\mathbf{W}}_\mathbf{k} + \tilde{\Delta}\mathbf{W_k})(\hat{\mathbf{Y}}_\mathbf{k} + \tilde{\Delta}\mathbf{Y_k})^\top\mathbf{b_j}),$$

where $\tilde{\Delta}\mathbf{W_k}$ and $\tilde{\Delta}\mathbf{Y_k}$ each represent the backward error for a matrix-vector multiply with inner products of lengths $m-(k-1)r$ and r in level-3 BLAS operations. If we let $|\tilde{\mathbf{\Delta}}\mathbf{W_k}| \leq \tilde{\gamma}_{\mathbf{W_k}}|\hat{\mathbf{W}}_\mathbf{k}|$ and $|\tilde{\Delta}\mathbf{Y_k}| \leq \tilde{\gamma}_{\mathbf{Y_k}}|\hat{\mathbf{Y}}_\mathbf{k}|$, then we have

$$|\mathrm{fl}(\hat{\mathbf{X}}_\mathbf{k}\mathbf{b_j}) - \mathbf{X_k}\mathbf{b_j}| \leq \tilde{\gamma}_{\mathbf{X_k}}\left(|\mathbf{b_j}| + |\hat{\mathbf{W}}_\mathbf{k}||\hat{\mathbf{Y}}_\mathbf{k}|^\top|\mathbf{b_j}|\right),$$

where $\tilde{\gamma}_{\mathbf{X_k}}$ accounts for the error caused by perturbations $\tilde{\mathbf{\Delta}}\mathbf{W_k} + \mathbf{\Delta W_k}$, $\tilde{\mathbf{\Delta}}\mathbf{Y_k} + \mathbf{\Delta Y_k}$, and δ. In uniform precision, this is largely derived from

$$\begin{aligned}
&|(\mathbf{W_k} + \mathbf{\Delta W_k} + \tilde{\mathbf{\Delta}}\mathbf{W_k})(\mathbf{Y_k} + \mathbf{\Delta Y_k} + \tilde{\mathbf{\Delta}}\mathbf{Y_k})^\top - \mathbf{W_k}\mathbf{Y_k}^\top| \\
&\qquad\qquad \leq \left[(1+\gamma_r + r\tilde{\gamma}_{m-(k-1)r})(1+\gamma_{m-(k-1)r} + \tilde{\gamma}_{m-(k-1)r}) - 1\right]|\mathbf{W_k}||\mathbf{Y_k}|^\top \\
&\qquad\qquad \leq r\tilde{\gamma}_{m-(k-1)r}|\mathbf{W_k}||\mathbf{Y_k}|^\top,
\end{aligned}$$

since the subtraction step only adds a single rounding error. Note that we implicitly covered the same step of applying an WY update in the construction of $\mathbf{z_k}^{(j)}$, but used Lemma 3.4 instead since we were concerned with the error occurred at a single update. We conclude with a general bound,

$$fl(\hat{\mathbf{X}}_\mathbf{k}\mathbf{b_j}) = (\mathbf{X_k} + \mathbf{\Delta X_k})\mathbf{b_j}, \quad \|\mathbf{\Delta X_k}\|_F \leq \tilde{\gamma}_{\mathbf{X_k}}, \tag{3.15}$$

where $\tilde{\gamma}_{\mathbf{X_k}} = r\tilde{\gamma}_{m-(k-1)r}$ in uniform precision. A normwise bound for a general matrix-matrix multiplication operation is stated in section 19.5 of [12].

Multiple WY updates: line 8-9 of alg. 5. All that remains is to consider the application of successive WY updates to form the QR factorization computed with BQR denoted as $\mathbf{Q}_{BQR}$ and $\mathbf{R}_{BQR}$. We can apply Lemma 3.3 directly by setting $\mathbf{X_k} := \mathbf{I} - \mathbf{W_k}\mathbf{Y_k}^\top$ and consider the backward errors for applying the sequence to a vector, $\mathbf{x} \in \mathbb{R}^m$, as we did for Lemma 3.4. Since $\mathbf{X_k} = \mathbf{P_{(k-1)r+1}} \cdots \mathbf{P_{kr}}$, is simply a sequence of Householder transformations, it is orthogonal, i.e. $\|X_k\|_2 = 1$. We only need to replace with $\mathbf{x}$ with $\mathbf{A}[:, i]$'s to form the columnwise bounds for $\mathbf{R}_{BQR}$, and apply the transpose to $\hat{e}_i$'s to form the bounds for $\mathbf{Q}_{BQR}$. Then,

$$\left\|\prod_{k=1}^{N}(\mathbf{X_k} + \mathbf{\Delta X_k}) - \prod_{k=1}^{N}\mathbf{X_k}\right\|_F \leq \sum_{k=1}^{N}(1+\tilde{\gamma}_{\mathbf{X_k}}) - 1 \tag{3.16}$$

$$\leq \sum_{k=1}^{N}(1 + r\tilde{\gamma}_{m-(k-1)r}) - 1 \leq rN\tilde{\gamma}_m \equiv n\tilde{\gamma}_m. \tag{3.17}$$

We showed earlier in this section that HQR performed on $\mathbf{C_k}$ accrues componentwise error of order $\tilde{\gamma}_{m-(k-1)r}$ by applying Theorem 3.5, and the building of the W factor, $r\tilde{\gamma}_{m-(k-1)r}$ order error. Both of these are clearly small in comparison to the error from applying many WY transformations, so we attribute the leading order error to this last step shown in (3.17). The primary goal of the analysis presented in this section is to make the generalization to mixed-precision settings in section 4 easier, and readers should refer to [9, 12] for full details.

3.3. Block HQR with partitioned rows : Tall-and-Skinny QR (TSQR).

Some important problems that require QR factorizations of overdetermined systems include least squares problems, eigenvalue problems, low rank approximations, as well as other matrix decompositions. Although Tall-and-Skinny QR (TSQR) broadly refers to block QR factorization methods with row partitions, we will discuss a specific variant of TSQR which is also known as the AllReduce algorithm [18]. In this paper, the TSQR/AllReduce algorithm refers to the most parallel variant of the block QR factorization algorithms discussed in [8]. A detailed description and rounding error analysis of this algorithm can be found in [18], and we present a pseudocode for the algorithm in alg. 6. Our initial interest in this algorithm came from its parallelizable nature, which is particularly suitable to implementation on GPUs. Additionally, our numerical simulations (discussed in section 5) show that TSQR can not only increase the speed but also outperform the traditional HQR factorization in low precisions.

3.3.1. TSQR/AllReduce Algorithm.

Algorithm 6 partitions the rows of a tall-and-skinny matrix, $\mathbf{A}$. HQR is performed on each of those blocks and pairs of $\mathbf{R}$ factors are combined to form the next set of $\mathbf{A}$ matrices to be QR factorized. This process is repeated until only a single $\mathbf{R}$ factor remains, and the $\mathbf{Q}$ factor is built from all of the Householder constants and vectors stored at each level. The most gains from parallelization can be made in the initial level where the maximum number of independent HQR factorizations occur. Although more than one configuration of this algorithm may be available for a given tall-and-skinny matrix, the number of nodes available and the shape of the matrix eliminate some of those choices. For example, a 1600-by-100 matrix can be partitioned into 2, 4, 8, or 16 initial row-blocks but may be restricted by a machine with only 4 nodes, and a 1600-by-700 matrix can only be partitioned into 2 initial blocks. Our numerical experiments show that the choice in the initial partition, which directly relates to the recursion depth of TSQR, has an impact in the accuracy of the QR factorization.

We refer to *level* as the number of recursions in a particular TSQR implementation. An L-level TSQR algorithm partitions the original matrix into 2^L submatrices in the initial or 0^{th} level of the algorithm, and 2^{L-i} QR factorizations are performed in level i for $i = 1, \cdots, L$. The set of matrices that are QR factorized at each level i are called $\mathbf{A}_j^{(i)}$ for $j = 1, \cdots, 2^{L-i}$, where superscript (i) corresponds to the level and the subscript j indexes the row-blocks within level i. In the following sections, alg. 6 (`tsqr`) will find a TSQR factorization of a matrix $A \in \mathbb{R}^{m\times n}$ where $m \gg n$. The inline function `qr` refers to alg. 3 and we use alg. 2 as a subroutine of `qr`.

TSQR Notation. We introduce new notation due to the multi-level nature of the TSQR algorithm. In the final task of constructing $\mathbf{Q}$, $\mathbf{Q}_j^{(i)}$ factors are aggregated from each block at each level. Each $\mathbf{Q}_j^{(i)}$ factor from level i is partitioned such that two corresponding $\mathbf{Q}^{(i-1)}$ factors from level $i-1$ can be applied to them. The partition (approximately) splits $\mathbf{Q}_j^{(i)}$ into two halves, $[\tilde{\mathbf{Q}}_{j,1}^{(i)\top}\tilde{\mathbf{Q}}_{j,2}^{(i)\top}]^\top$. The functions $\alpha(j)$ and $\phi(j)$ are defined such that $\mathbf{Q}_j^{(i)}$ is applied to the correct blocks from the level below: $\tilde{\mathbf{Q}}^{(i+1)}_{\alpha(j),\phi(j)}$. For $j = 1, \cdots, 2^{L-i}$ at level i, we need $j = 2(\alpha(j) - 1) + \phi(j)$, where $\alpha(j) = \lceil \frac{j}{2} \rceil$ and $\phi(j) = 2 + j - 2\alpha(j) \in \{1, 2\}$. section 3.3.2 shows full linear algebra details for a single-level ($L = 1$, 2 initial blocks) example. The reconstruction of $\mathbf{Q}$ can be implemented more efficiently (see [3]), but the reconstruction method in alg. 6 is presented for a clear, straightforward explanation.

3.3.2. Single-level Example. In the single-level version of this algorithm, we first bisect $\mathbf{A}$ into $\mathbf{A}_1^{(0)}$ and $\mathbf{A}_2^{(0)}$ and compute the QR factorization of each of those submatrices. We combine the resulting upper-triangular matrices (see below) which is QR factorized, and the process is repeated:

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_1^{(0)}\\ \mathbf{A}_2^{(0)}\end{bmatrix} = \begin{bmatrix}\mathbf{Q}_1^{(0)}\mathbf{R}_1^{(0)}\\ \mathbf{Q}_2^{(0)}\mathbf{R}_2^{(0)}\end{bmatrix} = \begin{bmatrix}\mathbf{Q}_1^{(0)} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_2^{(0)}\end{bmatrix}\begin{bmatrix}\mathbf{R}_1^{(0)}\\ \mathbf{R}_2^{(0)}\end{bmatrix} = \begin{bmatrix}\mathbf{Q}_1^{(0)} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_2^{(0)}\end{bmatrix}\mathbf{A}_1^{(1)} = \begin{bmatrix}\mathbf{Q}_1^{(0)} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_2^{(0)}\end{bmatrix}\mathbf{Q}_1^{(1)}\mathbf{R}.$$

The $\mathbf{R}$ factor of $\mathbf{A}_1^{(1)}$ is the final $\mathbf{R}$ factor of the QR factorization of the original matrix, $\mathbf{A}$. However, the final $\mathbf{Q}$ still needs to be constructed. Bisecting $\mathbf{Q}_1^{(1)}$ into two submatrices, i.e. $\tilde{\mathbf{Q}}_{1,1}^{(1)}$ and $\tilde{\mathbf{Q}}_{1,2}^{(1)}$, allows us to write and compute the product more compactly,

$$\mathbf{Q} := \begin{bmatrix}\mathbf{Q}_1^{(0)} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_2^{(0)}\end{bmatrix}\mathbf{Q}_1^{(1)} = \begin{bmatrix}\mathbf{Q}_1^{(0)} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_2^{(0)}\end{bmatrix}\begin{bmatrix}\tilde{\mathbf{Q}}_{1,1}^{(1)}\\ \tilde{\mathbf{Q}}_{1,2}^{(1)}\end{bmatrix} = \begin{bmatrix}\mathbf{Q}_1^{(0)}\tilde{\mathbf{Q}}_{1,1}^{(1)}\\ \mathbf{Q}_2^{(0)}\tilde{\mathbf{Q}}_{1,2}^{(1)}\end{bmatrix}.$$

More generally, alg. 6 takes a tall-and-skinny matrix $\mathbf{A}$ and level L and finds a QR factorization by initially partitioning $\mathbf{A}$ into 2^L row-blocks and includes the building of $\mathbf{Q}$. For simplicity, we assume that m is exactly $h2^L$ so that the initial partition yields 2^L blocks of equal sizes, h-by-n. Also, note that `hh_mult` refers to the action of applying multiple Householder transformations given a set of Householder vectors and constants, which can be performed by iterating line 6 of alg. 3. This step can be done in a level-3 BLAS operation via a WY update if alg. 6 was modified to store the WY representation at the QR factorization of each block of each level, $\mathbf{A}_j^{(i)}$.

3.3.3. TSQR: Rounding Error Analysis. The TSQR algorithm presented in alg. 6 is a divide-and-conquer strategy for the QR factorization that uses the HQR within the subproblems. Divide-and-conquer methods can naturally be implemented in parallel and accumulate less rounding errors. For example, the single-level TSQR decomposition of a tall-and-skinny matrix, $\mathbf{A}$ requires 3 total HQRs of matrices of sizes $\lfloor \log_2(\frac{m}{n})\rfloor$-by-$n$, $\lceil \log_2(\frac{m}{n})\rceil$-by-$n$, and $2n$-by-n. The single-level TSQR strictly uses more FLOPs, but the dot product subroutines may accumulate smaller rounding errors (and certainly have smaller upper bounds) since they are performed on shorter vectors, and lead to a more accurate solution overall. These concepts are elucidated in [18], where the rounding error analysis of TSQR is shown in detail in [18]. We summarize the main results from [18] in Theorem 3.8.

Algorithm 6: $\mathbf{Q}, \mathbf{R} = \texttt{tsqr}(\mathbf{A}, L)$. Finds a QR factorization of a tall, skinny matrix, $\mathbf{A}$.

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$ where $m \gg n$, $L \leq \lfloor \log_2\left(\frac{m}{n}\right)\rfloor$, and 2^L is the initial number of blocks.
Output: $\mathbf{Q} \in \mathbb{R}^{m\times n}$, $\mathbf{R} \in \mathbb{R}^{n\times n}$ such that $\mathbf{QR} = \mathbf{A}$.

$h \leftarrow \lfloor \frac{m}{2^L} \rfloor$ `// Number of rows.`
`/* Split` $\mathbf{A}$ `into` 2^L `blocks. Note that level` (i) `has` 2^{L-i} `blocks. */`
for $j = 1 : 2^L$ **do**
  $\mathbf{A}_j^{(0)} \leftarrow \mathbf{A}[(j-1)h+1 : jh, :]$
`/* Store Householder vectors as columns of matrix` $\mathbf{V}_j^{(i)}$`, Householder constants as components of vector` $\boldsymbol{\beta}_j^{(i)}$`, and set up the next level. */`
for $i = 0 : L-1$ **do**
  `/* The inner loop can be parallelized. */`
  for $j = 1 : 2^{L-i}$ **do**
    $\mathbf{V}_{2j-1}^{(i)}, \boldsymbol{\beta}_{2j-1}^{(i)}, \mathbf{R}_{2j-1}^{(i)} \leftarrow \texttt{qr}(\mathbf{A}_{2j-1}^{(i)})$
    $\mathbf{V}_{2j}^{(i)}, \boldsymbol{\beta}_{2j}^{(i)}, \mathbf{R}_{2j}^{(i)} \leftarrow \texttt{qr}(\mathbf{A}_{2j}^{(i)})$
    $\mathbf{A}_j^{(i+1)} \leftarrow \begin{bmatrix} \mathbf{R}_{2j-1}^{(i)} \\ \mathbf{R}_{2j}^{(i)} \end{bmatrix}$
$\mathbf{V}_1^{(L)}, \boldsymbol{\beta}_1^{(L)}, \mathbf{R} \leftarrow \texttt{qr}(\mathbf{A}_1^{(L)})$ `// The final` $\mathbf{R}$ `factor is built.`
$\mathbf{Q}_1^{(L)} \leftarrow \texttt{hh\_mult}(\mathbf{V}_1^{(L)}, I_{2n\times n})$
`/* Compute` $\mathbf{Q}^{(i)}$ `factors by applying` $\mathbf{V}^{(i)}$ `to` $\mathbf{Q}^{(i+1)}$ `factors. */`
for $i = L-1 : -1 : 1$ **do**
  for $j = 1 : 2^{L-i}$ **do**
    $\mathbf{Q}_j^{(i)} \leftarrow \texttt{hh\_mult}\left(\mathbf{V}_j^{(i)}, \begin{bmatrix} \tilde{\mathbf{Q}}_{\alpha(j),\phi(j)}^{(i+1)} \\ \mathbf{0} \end{bmatrix}\right)$
$\mathbf{Q} \leftarrow [];$ `// Construct the final` $\mathbf{Q}$ `factor.`
for $j = 1 : 2^L$ **do**
  $\mathbf{Q} \leftarrow \begin{bmatrix} \mathbf{Q} \\ \texttt{hh\_mult}\left(\mathbf{V}_j^{(0)}, \begin{bmatrix} \tilde{\mathbf{Q}}_{\alpha(j),\phi(j)}^{(1)} \\ \mathbf{0} \end{bmatrix}\right) \end{bmatrix}$
return $\mathbf{Q}$, $\mathbf{R}$

THEOREM 3.8. *Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$ have full rank, n, and $\hat{\mathbf{Q}} \in \mathbb{R}^{m\times n}$ and $\hat{\mathbf{R}} \in \mathbb{R}^{n\times n}$ be the thin QR factors of $\mathbf{A}$ obtained via alg. 6 with L levels. Let us further assume that $m = h2^L$. Then we have normwise forward error bounds*

$$\hat{\mathbf{A}} = \mathbf{A} + \boldsymbol{\Delta}\mathbf{A} = \mathbf{Q}(\mathbf{R} + \boldsymbol{\Delta}\mathbf{R}),$$
$$\hat{\mathbf{Q}} = \mathbf{Q} + \boldsymbol{\Delta}\mathbf{Q},$$

where

$$\|\mathbf{\Delta R}\|_F, \|\mathbf{\Delta A}\|_F \leq \left[n\tilde{\gamma}_h + (1+n\tilde{\gamma}_h)\left\{(1+n\tilde{\gamma}_{2n})^L - 1\right\}\right]\|\mathbf{A}\|_F, \text{ and} \tag{3.18}$$

$$\|\mathbf{\Delta Q}\|_F \leq \sqrt{n}\left[(1+n\tilde{\gamma}_h)(1+n\tilde{\gamma}_{2n})^L - 1\right]. \tag{3.19}$$

Furthermore, if we assume $n\tilde{\gamma}_h, n\tilde{\gamma}_{2n} \ll 1$, the coefficient for $\|\mathbf{A}\|_F$ in (3.18) can be approximated as

$$\left[n\tilde{\gamma}_h + (1+n\tilde{\gamma}_h)\left\{(1+n\tilde{\gamma}_{2n})^L - 1\right\}\right] \simeq n\tilde{\gamma}_h + Ln\tilde{\gamma}_{2n}, \tag{3.20}$$

and the right hand side of (3.19) can be approximated as

$$\sqrt{n}\left[(1+n\tilde{\gamma}_h)(1+n\tilde{\gamma}_{2n})^L - 1\right] \simeq \sqrt{n}\left(n\tilde{\gamma}_h + Ln\tilde{\gamma}_{2n}\right). \tag{3.21}$$

We can also form a backward error, where $\mathbf{A}+\mathbf{\Delta A}_{TSQR} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$, and both $\hat{\mathbf{Q}}$ and $\hat{\mathbf{R}}$ are obtained via alg. 6. Then,

$$\|\mathbf{\Delta A}_{TSQR}\|_F = \|\mathbf{Q}\mathbf{\Delta R} + \mathbf{\Delta Q}\hat{\mathbf{R}}\|_F \simeq \sqrt{n}\left(n\tilde{\gamma}_h + Ln\tilde{\gamma}_{2n}\right)\|\mathbf{A}\|_F. \tag{3.22}$$

Note that the $n\tilde{\gamma}_h$ and $n\tilde{\gamma}_{2n}$ terms correspond to errors from applying HQR to the blocks in the initial partition and to the blocks in levels 1 through L respectively. We can easily replace these with analogous mixed-precision terms and keep the analysis accurate. Both level-2 and level-3 BLAS implementations will be considered in section 4.

4. Mixed-precision error analysis.

Let us first consider rounding errors incurred from carrying out HQR in high precision, then cast down at the very end. This could be useful in applications that require economical storage, but have enough memory to carry out HQR in higher precision. Suppose two types of floating point types $\mathbb{F}_{low}$ and $\mathbb{F}_{high}$ where $\mathbb{F}_{low} \subseteq \mathbb{F}_{high}$, and for all $x, y \in \mathbb{F}_{low}$, the exact product xy can be represented in $\mathbb{F}_{high}$. Some example pairs of $\{\mathbb{F}_{low}, \mathbb{F}_{high}\}$ include $\{\text{fp16}, \text{fp32}\}$, $\{\text{fp32}, \text{fp64}\}$, and $\{\text{fp16}, \text{fp64}\}$. Suppose that the matrix to be factorized is stored with low precision numbers, $\mathbf{A} \in \mathbb{F}_{low}^{m\times n}$. Casting up adds no rounding errors, so we can directly apply the analysis that culminated in Theorem 3.5, and we only consider the columnwise forward error in the $\mathbf{Q}$ factor.

Then, the j^{th} column of $\hat{\mathbf{Q}}_{HQR}$ is bounded normwise via

$$\|\mathbf{Q}[:,j]\|_2 \leq n\tilde{\gamma}_{high},$$

and incurs an extra rounding error when $\mathbf{Q} \in \mathbb{F}_{high}^{m\times n}$ is cast down to $F_{low}^{m\times n}$. Since $\hat{\mathbf{Q}}_{HQR}$ should be almost orthogonal with respect to the higher precision, we can expect all components to be within the dynamic range of $\mathbb{F}_{low}$.

In the next sections, we consider performing BQR and TSQR with FLOPs within a block and/or a level in high precision, but cast down to low precision in between blocks in 4.1. Finally, we consider all 3 algorithms with an ad hoc mixed-precision setting where inner products are performed in high precision and all other operations are computed in low precision in 4.2.

4.1. Round down at block-level (BLAS-3).

4.2. Round down at inner-product level (BLAS-2).

5. Numerical Experiments.

6. Conclusion. Though the use of lower precision naturally reduces the bandwidth and storage needs, the development of GPUs to optimize low precision floating point arithmetic have accelerated the interest in half precision and mixed-precision algorithms. Loss in precision, stability, and representable range offset for those advantages, but these shortcomings may have little to no impact in some applications. It may even be possible to navigate around those drawbacks with algorithmic design.

The existing rounding error analysis cannot accurately bound the behavior of mixed-precision arithmetic. We have developed a new framework for mixed-precision rounding error analysis and applied it to HQR, a widely used linear algebra routine, and implemented it in an iterative eigensolver in the context of spectral clustering. The mixed-precision error analysis builds from the inner product routine, which can be applied to many other linear algebra tools as well. The new error bounds more accurately describe how rounding errors are accumulated in mixed-precision settings. We also found that TSQR, a communication-avoiding, easily parallelizable QR factorization algorithm for tall-and-skinny matrices, can outperform HQR in mixed-precision settings for ill-conditioned, extremely overdetermined cases, which suggests that some algorithms are more robust against lower precision arithmetic.

Although this work is focused on QR factorizations and applications in spectral clustering, the mixed precision round-off error analysis can be applied to other tasks and applications that can benefit from employing low precision computations. While the emergence of technology that support low precision floats combats issues dealing with storage, now we need to consider how low precision affects stability of numerical algorithms.

Future work is needed to test larger, more ill-conditioned problems with different mixed-precision settings, and to explore other divide-and-conquer methods like TSQR that can harness parallel capabilities of GPUs while withstanding lower precisions.

REFERENCES

[1] A. Abdelfattah, S. Tomov, and J. Dongarra, *Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs*, in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2019, pp. 111–122, https://doi.org/10.1109/IPDPS.2019.00022.

[2] J. Appleyard and S. Yokim, *Programming Tensor Cores in CUDA 9*, 2017, https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/ (accessed 2018-07-30).

[3] G. Ballard, J. W. Demmel, L. Grigori, M. Jacquelin, H. Diep Nguyen, and E. Solomonik, *Reconstructing Householder vectors from tall-skinny QR*, vol. 85, 05 2014, pp. 1159–1170, https://doi.org/10.1109/IPDPS.2014.120.

[4] C. Bischof and C. Van Loan, *The WY Representation for Products of Householder Matrices*, SIAM Journal on Scientific and Statistical Computing, 8 (1987), pp. s2–s13, https://doi.org/10.1137/0908009.

[5] M. Courbariaux, Y. Bengio, and J.-P. David, *Training deep neural networks with low precision multiplications*, arXiv preprint, arXiv:1412.7024, (2014).

[6] M. Courbariaux, J.-P. David, and Y. Bengio, *Low precision storage for deep learning*, arXiv preprint arXiv:1412.7024, (2014).

[7] J. Demmel, I. Dumitriu, and O. Holtz, *Fast linear algebra is stable*, Numerische Mathematik, 108 (2007), pp. 59–91, https://doi.org/10.1007/s00211-007-0114-x, https://arxiv.org/abs/0612264.

[8] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal on Scientific Computing, 34 (2012), https://doi.org/10.1137/080731992, https://arxiv.org/abs/0808.2664.

[9] G. H. Golub and C. F. Van Loan, *Matrix computations*, JHU press, 4 ed., 2013.

[10] A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov, and J. Dongarra, *The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques*, June 2018, pp. 586–600, https://doi.org/10.1007/978-3-319-93698-7_45.

[11] 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 Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, Piscataway, NJ, USA, 2018, IEEE Press, pp. 47:1–47:11, https://doi.org/10.1109/SC.2018.00050, https://doi.org/10.1109/SC.2018.00050.

[12] N. J. Higham, *Accuracy and Stability of Numerical Methods*, 2002, https://doi.org/10.2307/2669725.

[13] N. J. Higham and T. Mary, *A New Approach to Probabilistic Rounding Error Analysis*, SIAM Journal on Scientific Computing, 41 (2019), pp. A2815–A2835, https://doi.org/10.1137/18M1226312, https://epubs.siam.org/doi/10.1137/18M1226312.

[14] N. J. Higham and S. Pranesh, *Simulating Low Precision Floating-Point Arithmetic*, SIAM Journal on Scientific Computing, 41 (2019), pp. C585–C602, https://doi.org/10.1137/19M1251308, https://epubs.siam.org/doi/10.1137/19M1251308.

[15] A. S. Householder, *Unitary triangularization of a nonsymmetric matrix*, Journal of the ACM (JACM), 5 (1958), pp. 339–342.

[16] I. C. F. Ipsen and H. Zhou, *Probabilistic Error Analysis for Inner Products*, (2019), http://arxiv.org/abs/1906.10465, https://arxiv.org/abs/1906.10465.

[17] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, *Mixed precision training*, in International Conference on Learning Representations, 2018, https://openreview.net/forum?id=r1gs9JgRZ.

[18] D. Mori, Y. Yamamoto, and S. L. Zhang, *Backward error analysis of the AllReduce algorithm for householder QR decomposition*, Japan Journal of Industrial and Applied Mathematics, 29 (2012), pp. 111–130, https://doi.org/10.1007/s13160-011-0053-x.

[19] R. Schreiber and C. Van Loan, *A Storage-Efficient WY Representation for Products of Householder Transformations*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 53–57, https://doi.org/10.1137/0910005.

[20] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin, *A transprecision floating-point platform for ultra-low power computing*, in 2018 Design, Automation Test in Europe Conference Exhibition (DATE), March 2018, pp. 1051–1056, https://doi.org/10.23919/DATE.2018.8342167.