Mixed Precision Randomized Low-Rank Approximation with GPU Tensor Cores

Marc Baboulin[1], Simplice Donfack[2,5], Oguz Kaya[3], Theo Mary[4], and Matthieu Robeyns[3(✉)]

[1] Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France
marc.baboulin@universite-paris-saclay.fr
[2] Université Paris-Saclay, UVSQ, Inria, CNRS, CEA, Maison de la Simulation, Gif-sur-Yvette, France
simplice.donfack@cea.fr
[3] Université Paris-Saclay, CNRS, LISN, Orsay, France
{oguz.kaya,matthieu.robeyns}@universite-paris-saclay.fr
[4] Sorbonne Université, CNRS, LIP6, Paris, France
theo.mary@lip6.fr
[5] Institut du Développement et des Ressources en Informatique Scientifique, Rue John von Neumann, 91403 Orsay, France

Abstract. Randomized projection methods have been shown to be very efficient at computing low-rank approximations (LRA) of large matrices. In this work, we investigate the design and development of such methods capable of exploiting recent mixed precision accelerators like GPUs equipped with tensor core units. We combine three new ideas to exploit mixed precision arithmetic in randomized LRA. The first is to perform the matrix multiplication with mixed precision fp16/fp32 tensor cores. The second is to use CholeskyQR orthonormalization, which is much faster on GPUs, while mitigating its numerical instability by using fp64 arithmetic. The third is to use a recently proposed iterative refinement method for LRA to improve the accuracy of the LRA by calling it twice. We implement the proposed approach on various GPU architectures and analyze its performance and accuracy. We compare with a standard randomized LRA entirely in fp32 arithmetic, which achieves an average accuracy of order 10^{-4}. Our results show that our approach without refinement is up to 8× faster, with an average accuracy of order 10^{-2}, which may be acceptable for some applications. Otherwise, we show that using refinement significantly improves the accuracy to an average of order 10^{-5}, while remaining up to 2.2× faster than the standard fp32 randomized LRA. This work illustrates the convergence of approximate computing techniques by combining low-rank approximations, randomization, mixed precision arithmetic, and GPU acceleration.

Keywords: mixed precision algorithms · randomized algorithms · low-rank approximations · GPU computing · tensor cores

AMS subject classifications. 65F45 · 65F55 · 65Y05 · 65Y10

J. Carretero et al. (Eds.): Euro-Par 2024, LNCS 14803, pp. 31–44, 2024.
https://doi.org/10.1007/978-3-031-69583-4_3

1 Introduction

Random projection methods are simple and robust techniques for reducing the dimensionality of data while preserving its structure [6]. These methods are commonly used in machine learning and many other application domains for the flexible tradeoff they offer between accuracy and performance [13]. Moreover, the matrix operations at the heart of these methods make them highly suitable for exploiting accelerators such as GPUs [10].

This work is motivated by the recent technological advances that allow for large speedups by using low precision arithmetic [7]. In particular, NVIDIA GPUs are equipped with so-called tensor core units that can perform matrix multiplication in mixed precision fp16/fp32 arithmetic up to 16× faster than in standard fp32 arithmetic [3]. However, the ability to reliably and efficiently exploit these low precision accelerators in application codes strongly depends on both the ability of these codes to use matrix operations in low precision, and their tolerance for rounding errors. In this paper we investigate to what extent these very fast low precision units can be exploited for accelerating randomized projection methods.

Specifically, we consider a randomized algorithm for computing low-rank approximations (LRA) based on random Gaussian sampling [6, Alg. 4.1]. Most of the literature discussing randomized LRA methods and their implementation on GPUs has focused on either exact arithmetic or fixed precision arithmetic, one notable exception being the recent work of Ootomo and Yokota [11] that we discuss in the next section. The main contribution of this article is the design of a new mixed precision randomized LRA method, with a performance and accuracy analysis showing that the proposed method is able to exploit GPU tensor cores reliably and efficiently.

Our method is based on three key ideas:

- The first idea consists in *performing the matrix–matrix products (GEMM kernel) in mixed precision arithmetic using the tensor cores*, since these operations represent the asymptotic bottleneck of the method. We compare several GEMM variants depending on how the conversions between fp32 and fp16 are handled, and identify one variant in particular that achieves the best performance–accuracy tradeoff.
- Then, having significantly accelerated the GEMM operations, we observe that the orthonormalization step (QR kernel), despite requiring an asymptotically negligible number of flops, becomes the new performance bottleneck. Then the second idea is to *switch the orthonormalization method from the standard Householder QR to a CholeskyQR algorithm*, which mainly relies on GEMM and is therefore much more efficient on GPUs. We mitigate the inherent instability of CholeskyQR by performing it in fp64 rather than fp32 arithmetic.
- This leads to a mixed precision randomized LRA method employing three precisions (fp16, fp32, and fp64). We show that this method can be up to 8× faster than the standard randomized LRA method in fixed precision fp32 arithmetic and achieves an average accuracy of order 10^{-2}, which may be

sufficient for some applications. Then the third idea is to *use the iterative refinement method for LRA recently proposed by Baboulin et al.* [2] *to improve the accuracy of the method.* This method consists in repeating the randomized LRA a second time on an error matrix whose rank is twice the rank of the original matrix. We show that with refinement, the accuracy of this method improves significantly to an average of order 10^{-5}, while still being up to 2.2× faster than the standard LRA method in fp32 arithmetic.

The paper is structured as follows. In Sect. 2, we provide the necessary technical background on randomized LRA methods and discuss related work. In Sect. 3, we describe the proposed mixed precision method and its implementation using GPU tensor cores. In Sect. 4, we perform some experiments to analyze the performance and accuracy of our method. We finally provide our concluding remarks in Sect. 5.

2 Background

2.1 Randomized LRA

Given a matrix $A \in \mathbb{R}^{m\times n}$, we want to approximate A as a low-rank product XY^T of smaller matrices $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$, where the rank k is (much) smaller than $\min(m,n)$. Such a low-rank approximation (LRA) is a powerful tool to reduce the costs for storing A and performing computations with it.

Among the many possible methods to compute LRA, randomized ones have encountered much success due to their ability to mainly rely on efficient matrix–matrix products. In this article, we focus on randomized LRA based on Gaussian sampling [6], as outlined in Algorithm 1. This method generates a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times \ell}$ and projects the matrix A onto it by computing the matrix–matrix product $B = A\Omega$ (line 2). The dimension ℓ is equal to $k + p$, where p is a small oversampling parameter (typically, $p \leq 10$). Then, the matrix B is orthonormalized via a QR factorization (line 3), yielding a matrix Q that satisfies $A \approx QQ^TA$. We can therefore obtain a rank-k approximation of A by setting X and Y to the first k columns of Q and Q^TA, respectively (lines 5 and 6); the latter is computed via a second matrix–matrix product (line 4).

We note that many alternative variants of Algorithm 1 are possible; for example the sampling may be performed differently (e.g., via a fast Fourier transform), or we may compute specific types of LRA (e.g., SVD) by further decomposing Q^TA. In addition, while Algorithm 1 is a fixed-rank algorithm, fixed-accuracy variants have also been proposed [6,9], in which an accuracy threshold ε is prescribed and the rank k_ε is adaptively discovered by the algorithm. In this article, we focus on the simplest variant described here and refer to the extensive survey [6] for further options.

Algorithm 1 relies on two computational kernels: matrix–matrix products (GEMM kernel) and QR factorization (QR kernel). Importantly, the GEMM kernel performs $4mn\ell$ flops whereas the QR kernel only performs $c_{\text{qr}}m\ell^2$ flops (where c_{qr} is a small constant that depends on the specific QR factorization

Algorithm 1: Randomized low-rank approximation.

Input : $A \in \mathbb{R}^{m \times n}$, the target rank k, the oversampling p.
Output : $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$.

1 $\Omega \leftarrow \texttt{randn}(n, k+p)$
2 $B \leftarrow A\Omega$
3 $Q \leftarrow \texttt{qr}(B)$
4 $Y \leftarrow A^T Q$
5 $X = Q(:, 1:k)$
6 $Y = Y(:, 1:k)$

method that is used). Therefore, the performance of the overall method should be guided by the GEMM kernel, which can be performed very efficiently on modern computer architectures and especially on GPU accelerators.

2.2 Related Work on Mixed Precision Randomized LRA

It is natural to seek to exploit the high-speed low precision arithmetic available on such GPU hardware to accelerate randomized LRA. However, the literature on these methods has mainly focused on either exact arithmetic or fixed precision arithmetic, where all the operations are performed in the same precision. To the best of our knowledge, only three recent papers depart from an exact or fixed arithmetic context to propose mixed precision variants of randomized LRA: Connolly, Higham, and Pranesh [4], Ootomo and Yokota [11], and Baboulin et al. [2].

Connolly, Higham, and Pranesh [4] propose a mixed precision variant of the adaptive randomized SVD algorithm of Martinsson and Voronin [9]. This variant relies on the observation that the norm of the matrix deflated with the current LRA may rapidly decrease, which makes it possible to switch the computation to lower precision. This observation is linked to the decay of the singular values of the matrix, which is also exploited by Amestoy et al. [1]. In this article, we do not consider adaptive (fixed-accuracy) variants of randomized LRA and do not assume any decay of the singular values of the matrix.

Ootomo and Yokota [11] propose a mixed precision variant of fixed-rank randomized SVD, which is very similar to Algorithm 1 (with simply one extra step to compute the SVD of $Q^T A$). This variant relies on the observation that the random Gaussian matrix Ω can be represented in fp16 arithmetic without endangering the stability of the computation. As a result, the GEMM $B = A\Omega$ can be efficiently performed using GPU tensor cores by computing $B = A_1\Omega + A_2\Omega$, where $A \approx A_1 + A_2$ and both A_1 and A_2 are stored in fp16. This approach exploits multiword arithmetic to emulate fp32 arithmetic using fp16 computations, see also Fasi et al. [5] for further details about multiword arithmetic.

Baboulin et al. [2] propose a general iterative refinement framework for computing LRA that can be applied to any LRA method. We describe in Algorithm 2 the specialization of this iterative refinement method to the case where the LRA is computed via Algorithm 1. The idea is to first compute a rank-k approximation $A \approx X_1 Y_1^T$ by applying Algorithm 2 in low precision (line 1), then computing the error $E = A - X_1 Y_1^T$ in high precision (line 2), and finally reapplying Algorithm 2 on E to compute a rank-$2k$ approximation $E \approx X_2 Y_2^T$ (line 3). This is based on the observation that if A has rank approximately k, then E should have at most rank approximately $2k$. Hence, the final result of this method is a rank-$3k$ approximation XY^T given as the concatenation of the factors $[X_1, X_2]$ and $[Y_1, Y_2]$ (lines 4 and 5).

Algorithm 2: Randomized LRA with iterative refinement.

Input : $A \in \mathbb{R}^{m \times n}$, the target rank k, the oversampling p.
Output : $X \in \mathbb{R}^{m \times 3k}$ and $Y \in \mathbb{R}^{n \times 3k}$ such that $A \approx XY^T$.

1 $[X_1, Y_1] = \texttt{randLRA}(A, k, p)$; `// in low precision`
2 $E = A - X_1 Y_1^T$; `// in high precision`
3 $[X_2, Y_2] = \texttt{randLRA}(E, 2k, p)$; `// in low precision`
4 $X = [X_1, X_2]$
5 $Y = [Y_1, Y_2]$

Connolly et al. [4] and Baboulin et al. [2] are not concerned with the high performance implementation of their methods and only provide MATLAB experiments. Our method proposed in the next section and its GPU implementation should therefore be most directly compared to the method of Ootomo and Yokota [11]. As explained, their method does not reduce the accuracy of the LRA thanks to the emulation of fp32 arithmetic in the first GEMM and the use of fp32 arithmetic in the remaining kernels (QR and second GEMM). However, this limits the maximum speedup obtainable by this approach, since a large part of the computations is still executed in fp32 arithmetic; Ootomo and Yokota [11] thus report a speedup of 1.28× compared with randomized SVD entirely in fp32 arithmetic. In contrast, our approach is more performance-driven: we use tensor core arithmetic in both GEMM kernels (without emulating fp32 arithmetic), obtaining speedups of up to 8× at the price of a lesser accuracy; we then implement the iterative refinement method of Baboulin et al. [2] on GPUs to improve the accuracy while retaining speedups of up to 2.2×.

3 Mixed Precision Randomized LRA on GPU Tensor Cores

In this section, we propose a mixed precision variant of `randLRA`, the randomized LRA method outlined in Algorithm 1, and describe its GPU implementation. As

mentioned, `randLRA` relies primarily on two kernels: the matrix–matrix product (GEMM kernel) and the QR factorization (QR kernel).

3.1 GEMM Kernel

The GEMM kernel $C = AB$ can be executed up to 16× faster using GPU tensor core units. However, these units require the input matrices A and B to be represented in fp16 which necessitates a conversion when they are originally stored in fp32. We can distinguish several variants depending on which matrices are converted (A and B only, or also C), and on whether these conversions are handled explicitly or implicitly. Indeed, a first option is to handle conversions implicitly by keeping the matrices in the input fp32 format and letting cuBLAS itself perform the conversions to fp16; the advantage of this approach lies in the simplicity of the code and the efficiency of the conversion which is handled by the optimized library. In the explicit approach, on the other hand, we convert the input matrices to fp16 ourselves before calling the cuBLAS tensor core GEMM; even though our own conversion routine might be less efficient, the advantage of this approach is that matrices that were already converted to fp16 can be reused in other calls to tensor core GEMM without the need for further conversions (note that this however requires extra storage to store the explicitly converted matrix).

In summary, we evaluate three variants of the GEMM kernel. We denote these variants as `tgemm` (to indicate the use of tensor cores) and use the subscripts "32|32", "16|16", "16|32" to indicate the precision type of the input (A and B) and output (C). Note that if the input type is fp32, an implicit conversion to fp16 is performed, whereas, if the output type is fp32, no conversion to fp16 is required because tensor cores have the ability to accumulate directly in fp32 [3].

- $\texttt{tgemm}_{32|32}$: A, B, C are all stored in fp32; the GEMM implicitly converts A and B to fp16 during the computation but keeps C in fp32.
- $\texttt{tgemm}_{16|16}$: A, B, C are all explicitly converted from fp32 to fp16 before the computation of the GEMM, which does not need any conversions.
- $\texttt{tgemm}_{16|32}$: A and B are explicitly converted to fp16 but C is kept in fp32; the GEMM accumulates the computation in C in fp32 arithmetic and thus requires no further conversions.

We will compare the performance–accuracy tradeoff achieved by each of these three variants in our benchmarks in the next section.

3.2 QR Kernel

The second main kernel of `randLRA` is the QR factorization kernel. Several methods exist that achieve different tradeoffs between efficiency and stability, for example, classical or modified Gram–Schmidt, or Householder QR. The most stable approach is the Householder QR factorization, which is implemented in GPU libraries such as MAGMA and cuSOLVER. Unfortunately, Householder

QR is also inefficient on GPUs due to its limited parallelism, and current implementations do not exploit tensor core arithmetic. As a result, in the context of `randLRA` on GPU tensor cores, even though the GEMM kernel requires in theory an asymptotically dominant number of flops, in practice the QR kernel becomes the performance bottleneck. This is because the GEMM kernel strongly benefits from GPUs and especially mixed precision tensor core units, whereas the QR kernel is less efficient on GPUs and cannot exploit tensor core units.

Motivated by these observations, we propose to use instead the Cholesky QR factorization, a much faster variant of QR which mainly relies on matrix–matrix products and can thus exploit GPUs much more efficiently. Cholesky QR orthononormalizes a tall–skinny matrix A by computing the Gram matrix $B = A^T A$, computing its Cholesky factorization $R^T R = B$, and obtaining the orthonormal factor by the triangular solve $Q = AR^{-1}$. Unfortunately, the condition number of the Gram matrix $\kappa(B) = \kappa(A)^2$ is large even for moderately ill-conditioned A, which makes Cholesky QR unstable due to the possible breakdown of the Cholesky factorization of B if B is singular in the working precision.

In fp32 arithmetic such breakdowns are expected to occur when $\kappa(A) \gtrsim 10^4$. In order to address this issue, we switched the Cholesky QR factorization from fp32 to fp64 arithmetic; in this case, breakdowns can only occur when $\kappa(A) \gtrsim 10^8$. Since in the context of `randLRA` the input matrix is stored in fp32 arithmetic, breakdowns should thus never occur with Cholesky QR in fp64 arithmetic. The resulting algorithm is outlined in Algorithm 3. We note that the final triangular solution step (line 4) could be performed in fp32 without affecting stability [12]; this is an improvement that we will investigate in future work.

Algorithm 3: Cholesky QR kernel implementation on GPU.

Input : $A_{32} \in \mathbb{R}^{m \times n}$.
Output : Orthonormal factor $Q_{32} \in \mathbb{R}^{m \times n}$ of A_{32}.

1 $A_{64} = \texttt{fp64}(A_{32})$
2 $B_{64} = A_{64}^T A_{64}$
3 $R_{64} = \texttt{chol}(B_{64})$
4 $Q_{64} = A_{64} R_{64}^{-1}$
5 $Q_{32} = \texttt{fp32}(Q_{64})$

3.3 Randomized LRA

Having discussed the implementation of the GEMM and QR kernels on GPUs, we can now present the implementation of `randLRA`, outlined in Algorithm 4.

Depending on the GEMM variant used, `randLRA` takes different forms; in Algorithm 4, we describe the case where $\texttt{tgemm}_{16|32}$ is used, which allows for explicitly converting A to fp16 only once (line 3) and reusing it in both GEMM calls (lines 3 and 6). The QR kernel (line 4) can be either standard Householder QR in fp32 arithmetic, or the mixed precision Cholesky QR kernel presented previously (Algorithm 3).

Algorithm 4: Mixed precision `randLRA` on GPU tensor cores.

Input : $A_{32} \in \mathbb{R}^{m\times n}$, the target rank k, the oversampling p.
Output : $X_{16} \in \mathbb{R}^{m\times k}$ and $Y_{16} \in \mathbb{R}^{n\times k}$ such that $A_{32} \approx X_{16}Y_{16}^T$.

1 $\Omega_{16} = \texttt{randn}(n, k+p)$; // in fp16
2 $A_{16} = \texttt{fp16}(A_{32})$
3 $B_{32} = \texttt{tgemm}_{16|32}(A_{16}, \Omega_{16})$; // with fp16/fp32 tensor cores
4 $Q_{32} = \texttt{qr}(B_{32})$
5 $Q_{16} = \texttt{fp16}(Q_{32})$
6 $Y_{32} = \texttt{tgemm}_{16|32}(A_{16}^T, Q_{16})$; // with fp16/fp32 tensor cores
7 $X_{16} = \texttt{fp16}(Q_{32}(:, 1\colon k))$
8 $Y_{16} = \texttt{fp16}(Y_{32}(:, 1\colon k))$

Finally, we describe in Algorithm 5 our implementation of the iterative refinement approach of Baboulin et al. [2] on GPU tensor cores, using Algorithm 4 as the low precision `randLRA` method.

Algorithm 5: Mixed precision `randLRA` on GPU tensor cores, with iterative refinement.

Input : $A_{32} \in \mathbb{R}^{m\times n}$, the target rank k, the oversampling p.
Output : $X_{16} \in \mathbb{R}^{m\times 3k}$ and $Y_{16} \in \mathbb{R}^{n\times 3k}$ such that $A_{32} \approx X_{16}Y_{16}^T$.

1 $[X_{16}, Y_{16}] = \texttt{randLRA}(A_{32}, k, p)$
2 $E_{32} = A_{32} - \texttt{tgemm}_{16|32}(X_{16}, Y_{16}^T)$
3 $[X'_{16}, Y'_{16}] = \texttt{randLRA}(E_{32}, 2k, p)$
4 $X_{16} = [X_{16}, X'_{16}]$
5 $Y_{16} = [Y_{16}, Y'_{16}]$

4 Experiments

4.1 Experimental Setting

All experiments have been carried out on the Jean Zay supercomputer located at IDRIS[1] Each node is equipped with 2 Intel Cascade Lake 6240R processors and 8 NVIDIA A100 PCIe 40 GB GPUs, for a total memory of 768 GB. Although each node has several GPUs, we use a single GPU for these experiments. On both architectures, we use CUDA 12.0.0. The CUDA package provides access to the libraries cuBLAS, cuSOLVER and cuRAND.

The matrices used in our experiments are randomly generated. Given the specified rank k, we generate two random Gaussian matrices $X \in \mathbb{R}^{m\times k}$ and $Y \in$

[1] http://www.idris.fr/eng/jean-zay/index.html.

$\mathbb{R}^{n\times k}$ and define $A = XY^T$. In all experiments we do not use any oversampling ($p = 0$).

We measure the performance of our algorithms in TFLOPS (number of Tera floating-point operations per second), that is,

$$\text{Performance } = \frac{N_{\text{flops}}}{10^{12} \cdot \text{time}}.$$

To measure the "effective" performance of the algorithms, we use the same reference number of flops N_{flops} for all of them, regardless of their actual number of flops. Specifically we use

$$N_{\text{flops}} = 4mnk + 2nk^2 - \frac{2}{3}k^3 + O(mn),$$

which corresponds to the number of flops of the baseline version, which performs only two GEMMs and one QR factorization.

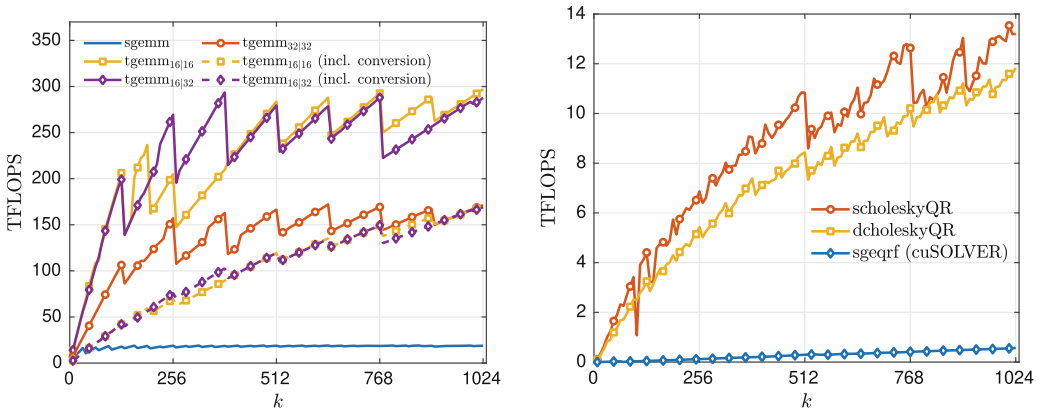
4.2 Performance and Accuracy of Kernels

In this section, we evaluate the performance of the building blocks of our `randLRA` algorithm: the GEMM and QR kernels.

GEMM Kernel. We begin by comparing in Fig. 1a the performance of the standard GEMM in fp32 arithmetic (`sgemm`) with the three `tgemm` variants that use the tensor cores described previously. The figure shows the performance for computing $C = AB$ where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times k}$, where $m = n = 35840$ are fixed and where k varies from 8 to 1024. (This shape of matrices corresponds to the two GEMMs performed by `randLRA`).

For the $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ variants, we plot their performance both with and without including the time taken by the explicit conversion of the matrices to fp16. For the $\texttt{tgemm}_{32|32}$ the implicit conversion is always performed and thus always included. The figure shows that, as expected, the implicit conversion performed by $\texttt{tgemm}_{32|32}$ is more efficient than the explicit one performed by our own implementation, so that this variant is faster than $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ if we include the time for explicit conversion. Interestingly, the relative cost of the conversion decreases as the dimension k increases, so that the performance of $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ including the conversion eventually becomes comparable to that of $\texttt{tgemm}_{32|32}$ for a sufficiently large k. More importantly, if we do not need to perform this conversion (because the input matrix is already stored in fp16), then the $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ variants become significantly faster than the $\texttt{tgemm}_{32|32}$ one.

We will investigate the difference in accuracy of these three variants directly in the context of their use in `randLRA`.



(a) GEMM (b) QR

Fig. 1. Performance of the GEMM and QR kernels.

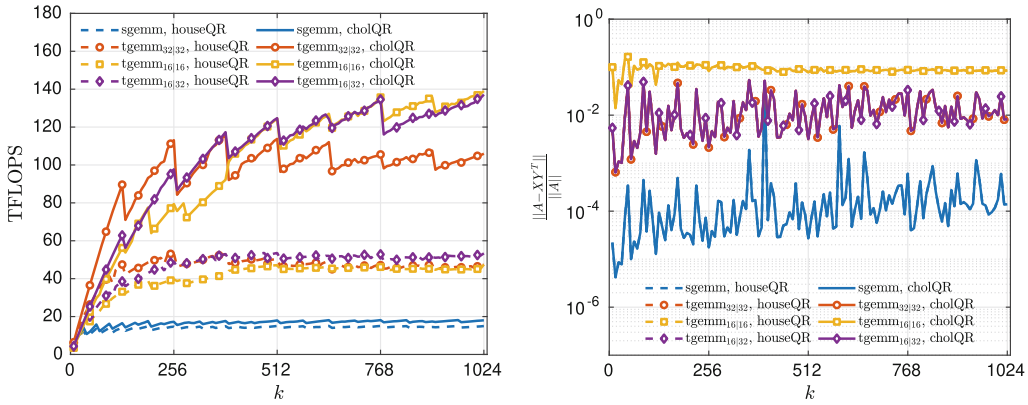
QR Kernel. We now turn to the performance of the QR kernel, reported in Fig. 1b. The figure compares the performance for orthonormalizing a matrix $B \in \mathbb{R}^{n \times k}$ where $n = 35840$ is fixed and where k varies from 8 to 1024. (Again, this corresponds to the shape of matrix arising in the QR kernel in `randLRA`).

We compare the classical Householder QR algorithm implemented in cuSOLVER using fp32 arithmetic (`sgeqrf`) with the Cholesky QR algorithm, using either fp32 or fp64 arithmetic (`scholQR` and `dcholQR`, the latter corresponding to Algorithm 3). At the time of these experiments, the only GPU implementation of Cholesky QR that we found is the one available in the MAGMA library. However, we found MAGMA's implementation not to be efficient for the target sizes in our context and therefore we made our own implementation.

Regarding the risk of Cholesky breaking down, in experiments with `randLRA` using Cholesky QR in fp32 (not shown), we found a significant fraction (about 14%) of breakdowns, which disappeared by using fp64 arithmetic instead. In comparison, Householder QR is robust (even in fp32 arithmetic), but extremely slow, as expected. Overall, our implementation of Cholesky QR in fp64 can be more than 20× faster than Householder QR in fp32.

4.3 Mixed Precision Randomized LRA

We now evaluate the performance and accuracy of randomized LRA, without iterative refinement to begin. We compare eight different variants. Two of them correspond to Algorithm 1 with the GEMM in standard fp32 arithmetic (`sgemm`), and with either fp32 Householder QR (`sgeqrf`) or fp64 Cholesky QR (`dcholQR`). The other six variants correspond to Algorithm 4, using one of the three `tgemm` variants for GEMM and again either fp32 Householder QR or fp64 Cholesky QR. Figure 2a plots the performance of these eight variants, and Fig. 2b plots their relative accuracy, measured as $\|A - XY^T\|/\|A\|$, where $\|\cdot\|$ denotes the Frobenius norm.



(a) Performance (b) Accuracy

Fig. 2. Performance and accuracy of `randLRA` *without* refinement. (In (b), $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ completely overlap both for Householder QR and Cholesky QR).

As a general preliminary observation, note that the use of Householder or Cholesky QR does not impact the accuracy for any of the variants (and so the latter is preferable since it is faster).

Let us first analyze the baseline variant using `sgemm` (fp32 arithmetic without tensor cores). This is the most accurate variant, with an average error of order 10^{-4}. However, this is also the slowest variant since it does not benefit from tensor cores: its performance is almost constant as soon as $k \geq 256$ and is limited to only 14 TFLOPS with Householder QR. The use of Cholesky QR slightly improves its performance but still remains limited to only 17 TFLOPS.

Then, let us now analyze the variants using `tgemm` (mixed precision GEMMs with tensor cores). Regardless of the choice of `tgemm` variant, using Householder QR limits the attainable performance to at best 50 TFLOPS. In this case, the GEMM performs well, but performance is limited by the performance of Householder QR, which is extremely slow as previously analyzed, and thus becomes the bottleneck, even though it requires an asymptotically negligible number of flops compared with GEMM. Therefore, for these `tgemm` variants, using Cholesky QR significantly improves performance by moving the bottleneck back to the GEMM.

Let us finally compare the three different `tgemm` variants to determine which is preferable. We can see in Fig. 2b that $\texttt{tgemm}_{16|16}$ is much less accurate than both $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$, with an average error of order 10^{-1} instead of 10^{-2}. This comes from C being stored in fp16: indeed, even though the tensor cores accumulate the operations in fp32 arithmetic, writing them in a matrix C stored in fp16 generates fp16 rounding errors which lead to a significant loss of accuracy. This effect has been well characterized in the literature, see in particular Blanchard et al. [3] for an error analysis and Lopez and Mary [8] for the consequence of this observation on LU factorization.

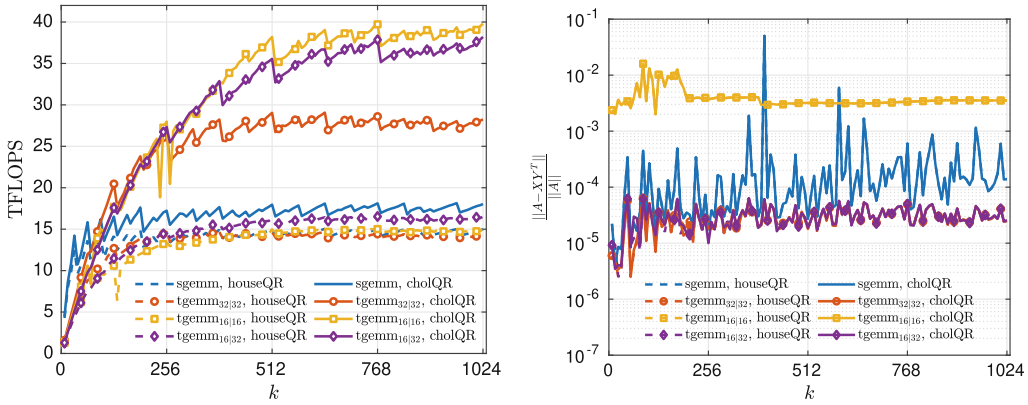
The choice of GEMM variant therefore comes down to which of $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ is faster. As Fig. 2a shows, this depends on the rank k. When k is small the relative cost of the conversion with respect to the computation is large and so $\texttt{tgemm}_{32|32}$ is faster than $\texttt{tgemm}_{16|32}$. As k increases the conversion becomes less and less costly with respect to the computation so eventually (here for $k \gtrsim 256$) $\texttt{tgemm}_{16|32}$ becomes faster, reaching up to 140 TFLOPS for large values of k. Therefore, `randLRA` and $\texttt{tgemm}_{16|32}$ is up to 8× faster than `randLRA` with `sgemm`, at the price of a lesser accuracy in this case without iterative refinement.

4.4 Iterative Refinement

Finally, we conclude by evaluating the performance and accuracy of `randLRA` using iterative refinement (IR). We compare the same eight variants as in the previous section, except that the variants that exploit mixed precision tensor cores (`tgemm`) now use IR (Algorithm 5); the two variants using `sgemm` *do not* use IR, since their accuracy is already satisfactory; we keep them as a reference point. Figure 3a plots the performance and Fig. 3b plots the relative accuracy.

In terms of accuracy, Fig. 3b confirms that the use of IR significantly improves the accuracy of all mixed precision variants. Specifically, the variants using $\texttt{tgemm}_{16|16}$ achieve an average error of order 10^{-2} instead of 10^{-1} and, more interestingly, the variants using $\texttt{tgemm}_{32|32}$ or $\texttt{tgemm}_{16|32}$ achieve an average error of order 10^{-5} instead of 10^{-2}. Thus, IR makes `randLRA` with these variants of `tgemm` at least as accurate, and in many cases even more accurate, than the standard `randLRA` with `sgemm`.

It remains to evaluate the impact of IR on performance. Note that the use of IR increases the number of flops by about a factor 4 (two GEMMs with rank k and three GEMMs with rank $2k$, instead of two GEMMs of rank k). For large values of k (for which the maximum performance is attained), the use of IR makes `randLRA` with $\texttt{tgemm}_{32|32}$ about 3.8× slower and `randLRA` with either $\texttt{tgemm}_{16|16}$ or $\texttt{tgemm}_{16|32}$ about 3.5× slower. The fact that the relative performance of $\texttt{tgemm}_{32|32}$ compared with $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ decreases when IR is used is explained by the fact that the relative weight of the conversions decreases with IR. In any case, the important conclusion is that the $\texttt{tgemm}_{16|32}$ variant with IR remains much faster than the variant with `sgemm` and no IR, with a speedup of up to a factor 2.2×. We therefore have obtained a method that is both faster *and* more accurate than the fp32 `randLRA` baseline.



(a) Performance (b) Accuracy

Fig. 3. Performance and accuracy of `randLRA` *with* refinement. (In (b), $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ completely overlap both for Householder QR and Cholesky QR).

5 Conclusion

We have proposed a new randomized low-rank approximation (LRA) method that efficiently and reliably exploits mixed precision GPUs. Our method, outlined in Algorithm 5, combines three key ideas. First, we use the GPU tensor core units to accelerate the matrix–matrix products (GEMM kernel) while minimizing the accuracy loss by using mixed precision arithmetic (matrices are converted to fp16 but computations are accumulated in fp32). Second, we replace the standard Householder QR by Cholesky QR, which is much more efficient on GPU, and we mitigate its inherent instability by performing it in fp64 arithmetic. Third and lastly, we implement the recently proposed iterative refinement approach for LRA [2] to recover full fp32 accuracy. Overall, our method achieves an accuracy that is at least as good and in many cases even better than a standard randomized LRA in fp32 arithmetic, while being up to 2.2× faster.

Our work gives rise to several perspectives. The first is to enhance the Cholesky QR algorithm using a mixed precision approach [12], which may in particular exploit the mixed precision tensor core units for the triangular solve while preserving the numerical stability. A second perspective concerns the design and implementation of efficient recompression methods to round the output matrix obtained by iterative refinement, which is of rank $3k$, back to the optimal rank k.

Finally our work illustrates the convergence of approximate computing techniques by combining low-rank approximations, randomization, mixed precision arithmetic, and GPU acceleration. Our results not only highlight the effectiveness of using mixed precision GPUs for accelerating randomized low-rank approximation while preserving a satisfying accuracy, but also pave the way toward exploring other similar GPU-based approximate methods in linear algebra and beyond.

Acknowledgments. This work was supported by the NumPEx Exa-SofT (ANR-22-EXNU-0003), MixHPC (ANR-23-CE46-0005-01), and SELESTE (ANR-20-CE46-0008-

01) projects of the French National Research Agency (ANR), and Paris Ile-de-France Region (DIM RFSI RC-TENSOR No 2021-05), and was performed using HPC resources from GENCI-IDRIS.

References

1. Amestoy, P., et al.: Mixed precision low rank approximations and their application to block low rank LU factorization. IMA J. Numer. Anal. **43**(4), 2198–2227 (2023). https://doi.org/10.1093/imanum/drac037
2. Baboulin, M., Kaya, O., Mary, T., Robeyns, M.: Mixed precision iterative refinement for low-rank matrix and tensor approximations (2023). https://inria.hal.science/hal-04115337
3. Blanchard, P., Higham, N.J., Lopez, F., Mary, T., Pranesh, S.: Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. SIAM J. Sci. Comput. **42**(3), C124–C141 (2020). https://doi.org/10.1137/19M1289546
4. Connolly, M.P., Higham, N.J., Pranesh, S.: Randomized low rank matrix approximation: rounding error analysis and a mixed precision algorithm. MIMS EPrint 2022.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK (2022). http://eprints.maths.manchester.ac.uk/2863/
5. Fasi, M., Higham, N.J., Lopez, F., Mary, T., Mikaitis, M.: Matrix multiplication in multiword arithmetic: error analysis and application to GPU tensor cores. J-SISC **45**(1), C1–C19 (2023). https://doi.org/10.1137/21m1465032
6. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev. **53**(2), 217–288 (2011). https://doi.org/10.1137/090771806
7. Higham, N.J., Mary, T.: Mixed precision algorithms in numerical linear algebra. Acta Numer. **31**, 347–414 (2022). https://doi.org/10.1017/s0962492922000022
8. Lopez, F., Mary, T.: Mixed precision LU factorization on GPU tensor cores: reducing data movement and memory footprint. Int. J. High Perfor. Comput. Appl. **37**(2), 165–179 (2023). https://doi.org/10.1177/10943420221136848
9. Martinsson, P.G., Voronin, S.: A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. SIAM J. Sci. Comput. **38**(5), S485–S507 (2016). https://doi.org/10.1137/15M1026080
10. Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., Dongarra, J.: Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. In: SC 2015 - International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, USA (2015). https://doi.org/10.1145/2807591.2807613
11. Ootomo, H., Yokota, R.: Mixed-Precision random projection for RandNLA on tensor cores. In: Proceedings of the Platform for Advanced Scientific Computing Conference, Association for Computing Machinery, New York, Davos, Switzerland (2023). https://doi.org/10.1145/3592979.3593413
12. Yamazaki, I., Tomov, S., Dongarra, J.: Mixed-precision cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. SIAM J. Sci. Comput. **37**(3), C307–C330 (2015). https://doi.org/10.1137/14M0973773
13. Yamazaki, I., Tomov, S., Dongarra, J.: Sampling algorithms to update truncated SVD. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 817–826 (2017). https://doi.org/10.1109/BigData.2017.8257997