Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs

Théo Mary
Université de Toulouse,
UPS-IRIT, France

Ichitaro Yamazaki
University of Tennessee,
Knoxville, Tennessee, U.S.A.

Jakub Kurzak
University of Tennessee,
Knoxville, Tennessee, U.S.A.

Piotr Luszczek
University of Tennessee,
Knoxville, Tennessee, U.S.A.

Stanimire Tomov
University of Tennessee,
Knoxville, Tennessee, U.S.A.

Jack Dongarra
University of Tennessee,
Knoxville, Tennessee, U.S.A.
Oak Ridge National Laboratory
University of Manchester

ABSTRACT

A low-rank matrix approximation plays an important role in many applications. To compute a low-rank approximation of a dense matrix, a common approach uses the QR factorization with column pivoting (QRCP). Though the reliability and efficiency of QRCP have been demonstrated, this deterministic approach requires costly communication at each step of the factorization. Since such communication is becoming increasingly expensive on modern computers, an alternative approach based on random sampling, which can be implemented using communication-optimal kernels, is becoming attractive. To study its potential, in this paper, we compare the performance of random sampling with that of QRCP on an NVIDIA Kepler GPU. Our performance results demonstrate that random sampling can be up to 13 times faster than the deterministic approach for computing the approximation of the same accuracy. We also present the parallel scaling of the random sampling over multiple GPUs, showing a speedup of 3.8 over three Kepler GPUs. These results demonstrate the potential of the random sampling as an excellent computational tool for many applications, and its potential is likely to grow on computers with a higher communication cost.

1. INTRODUCTION

A low-rank matrix approximation plays an important role in a number of areas of study which include theoretical computer science, numerical linear algebra, statistics, applied mathematics, data analysis, machine learning, and physical and biological sciences. In many cases, by taking advantages of their low-rank properties, we can reduce the computational and storage requirements of manipulating such matrices, or reduce the complexity of analyzing the given dataset. One standard algorithm to extract a low-rank approximation of a dense matrix A is based on the QR factorization with column pivoting (QRCP) [3]. After k steps of QRCP, we obtain a rank-k approximation of A:

$$\underset{m \times n}{AP} \approx \underset{m \times k}{Q} \underset{k \times n}{R}, \tag{1}$$

where Q has orthonormal columns, R is an upper triangular matrix, and the pivots P are selected to reveal the numerical rank of A. Though QRCP has been shown to be efficient and reliable in practice, this deterministic approach requires significant communication at each step of the factorization, where the communication includes the synchronization and the data transfer between the parallel processing units, as well as the data movement through the local memory hierarchy. In comparison to arithmetic operations, such communication is becoming increasingly expensive on the modern computers and even more so on the emerging computers – clearly, it is critical to consider this hardware trend when designing high-performance software.

To address these recent hardware trends, the algorithms based on random sampling have been gaining attention [11, 18, 14, 8, 12]. These algorithms first sample a subspace which approximates the range of the matrix A, and then extract the approximation of A from a low-rank approximation of the sampled matrix. The main reason for the increasing attention is that the sampled matrix can be computed in a communication-optimal fashion. In addition, the dimension of the sampled matrix is typically much smaller than the dimension of A and computing its low-rank approximation, even using a standard deterministic algorithm, requires only marginal computational and communication cost. As a result, compared to the deterministic approach, random sampling may better utilize the modern architecture, especially of a large-scale parallel computers with a high communication cost.

In this paper, we study the potential of the random sampling by first comparing its performance with that of QRCP on a GPU and then studying its parallel scaling on shared-memory multicore CPUs with multiple GPUs. Our performance results demonstrate that the random sampling can be up to 12.8 times faster than QRCP with one GPU, while

ICS'15 June 8-11, 2015, Newport Beach, CA, USA.

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

Notation	Description
$m \times n$	dimension of of the coefficient matrix A
k	target rank of A
p	oversampling dimension
ℓ	total sampling dimension (i.e. $\ell = k + p$)
q	number of power iterations
n_g	number of available GPUs
$A_{(i)}$	submatrix of A distributed on the i-th GPU
A_j and $A_{j_1:j_2}$	j-th and j_1-th through j_2-th columns of A

Figure 1: Notations used in this paper.

obtaining good parallel scaling over the multiple GPUs.

Main contributions

We outline the main contributions of this paper.

- We investigate the potential of random sampling on a GPU, namely when and how large performance improvement can be obtained over the traditional deterministic QRCP. Thus, this paper provides the insights on when the random sampling may be beneficial for the users.
- Although the algorithm is not novel, it is gaining attention to address many challenges on the emerging computers, and, to the best of our knowledge, we haven't found any detailed implementation or performance report of random sampling. Hence, our primary focus is to improve the robustness of the algorithm in practice (e.g., integrating power iterations and adaptive step sizes into the adaptive scheme) such that it can be used in many applications.
- Therefore, the paper describes our implementation in details (e.g., orthogonalization kernels), and although GPU kernel details were outside of the scope for this paper, the GPU kernels developed for these studies are state-of-the-art and will eventually be released as part of the numerical linear algebra package `MAGMA`[1].

The rest of the paper is organized as follows: first, in Sections 2 and 3, we review the QRCP and random sampling algorithms, respectively. Then, in Section 4, we describe our GPU implementations of the algorithms and its extension to utilize multiple GPUs. Next, in Section 5, we provide the performance model of the algorithms to discuss the potential of the algorithms in more general contexts. Finally, after discussing the experimental setups in Section 6, we present our numerical results, GPU kernel performance, and the performance of the random sampling with static and adaptive sampling sizes in Sections 7 through 10. We provide our final remarks in Section 11. Figure 1 lists the notations that will be used in the rest of this paper.

2. QR WITH COLUMN PIVOTING

To compute the low-rank approximation (1), the most-widely used algorithms are the variants of the QR factorization with column pivoting (QRCP) [3]. In this algorithm, the column with the largest norm is selected as a pivot at each step of the Householder QR factorization. Though the algorithm is not guaranteed to reveal the numerical rank, it is widely used because of its algorithmic simplicity and its efficiency and reliability in practice. In addition, it is possible to cheaply downdate the column norms at each step of the factorization, reducing its computational overhead over the standard Householder QR factorization.

A column-based QRCP uses BLAS-2 matrix-vector operations to update each column of A. On modern computers, the data movement is expensive, and the BLAS-2 kernels obtain only a small fraction of the hardware peak performance, limiting the performance of the column-based QRCP. To improve the data locality, a block-based QRCP [16] first factors a subset of the remaining columns of A (referred to as a panel), and then updates the trailing submatrix using the accumulated transformations at once. Since the trailing submatrix can be updated using BLAS-3 matrix-matrix operations, the block-based algorithm can exploit better data locality and obtain a higher performance. This BLAS-3 based QRCP is implemented in `LAPACK`[2], referred to as QP3, and widely used in practice. It is also possible to compute a truncated version of QP3 by returning after factoring the k columns of A.

Unfortunately, QP3 still performs about half of its floating-point operations (flops) using BLAS-2, and requires a synchronization to select a pivot at each step of the panel factorization. In addition, the round-off errors could accumulate and the downdated column norms could significantly diverge from the actual norms [16]. When this occurs, the trailing submatrix is immediately updated using the current Householder transformations and the column norms are recomputed. If the column norms need to be frequently recomputed, then the computational overhead could become significant. In addition, the frequent norm recomputation leads to poorer data locality since the column norms are computed using BLAS-1 vector-vector operations, and the trailing submatrix is updated using the smaller blocks.

3. RANDOM SAMPLING

Random sampling first samples a subspace that approximately spans the range of A and generates its orthogonal basis vectors $\widehat{Q}$. Then, the low-rank approximation of A is given by $A \approx B\widehat{Q}$, where $B = A\widehat{Q}^T$. It is also possible to compute the low-rank approximation of the form (1) based on the QRCP of the sampled matrix B. Specifically, the random sampling algorithm for computing (1) takes the following three steps:

1. Sampling (Step 1): Generate the sampled matrix B:
$$\underset{\ell \times n}{B} = \underset{\ell \times m}{\Omega}\ \underset{m \times n}{A},$$
where Ω is referred to as an $\ell \times m$ sampling matrix. A popular sampling matrix Ω includes a Gaussian random matrix and a FFT matrix [8].
2. QRCP (Step 2): Compute a QRCP factorization of the sampled matrix B:
$$\begin{aligned} BP &\approx \widehat{Q}\ \begin{pmatrix} \widehat{R}_{1:k} & \widehat{R}_{k+1:n} \end{pmatrix} \\ &= \widehat{Q}\widehat{R}_{1:k}\ \begin{pmatrix} I_k & \widehat{R}_{1:k}^{-1}\widehat{R}_{k+1:n} \end{pmatrix} \\ &= BP_{1:k}\ \begin{pmatrix} I_k & \widehat{R}_{1:k}^{-1}\widehat{R}_{k+1:n} \end{pmatrix}. \end{aligned}$$

[1] `http://icl.utk.edu/magma/`

[2] `http://www.netlib.org/lapack/`

Require: $m \times n$ matrix A.
1: ▷ Step 1: Gaussian sampling
2: $\Omega := \text{PRNG}(\ell, m)$, where $\ell = k + p$
3: $B := \Omega\, A$, $C := []$
4: $B := \text{POWER}(A, B, C, 1, \ell, q)$
5: ▷ Step 2: QRCP
6: $[\widehat{Q}, \widehat{R}, P] := \text{QRCP}(B)$
7: ▷ Step 3: QR
8: $[Q, \bar{R}] := \text{QR}(AP_{1:k})$
9: $T := \widehat{R}_{1:k}^{-1} \widehat{R}_{k+1:n}$
10: $R := \bar{R} \begin{pmatrix} I_k & T \end{pmatrix}$
11: **return** Q, R, P such that $AP \approx QR$.

1: ▷ $\text{POWER}(A, B, C, j, k, q)$
2: **for** 1, 2, ..., q **do**
3: ▷ Orthogonalization
4: $B_{j:k} := \text{BOrth}(B_{1:j-1}, B_{j:k})$
5: $B_{j:k} := \text{QR}(B_{j:k})$
6: ▷ Matrix-matrix multiply
7: $C_{j:k} := B_{j:k} A^T$
8: ▷ Orthogonalization
9: $C_{j:k} := \text{BOrth}(C_{1:j-1}, C_{j:k})$
10: $C_{j:k} := \text{QR}(C_{j:k})$
11: ▷ Matrix-matrix multiply
12: $B_{j:k} := C_{j:k} A$
13: **end for**
14: **return** B and C.

Figure 2: Random sampling algorithm, where $\text{PRNG}(\ell, m)$ returns an $\ell \times m$ Gaussian random matrix, $[Q, R] := \text{QR}(B)$ and $[Q, R, P] := \text{QRCP}(B)$ return the QR and QRCP factors of B (i.e., $QR = B$ and $BP = QR$), respectively, and $V := \text{BOrth}(B, Q)$ orthogonalizes B against Q (i.e., $V^T Q = 0$).

Thus, we have

$$AP \approx AP_{1:k} \begin{pmatrix} I_k & \widehat{R}_{1:k}^{-1} \widehat{R}_{k+1:n} \end{pmatrix}. \tag{2}$$

3. QR (Step 3): Compute the QR factorization of $AP_{1:k}$:

$$AP_{1:k} = Q\bar{R}. \tag{3}$$

Thus, combining (2) and (3), we obtain

$$\underset{m \times n}{AP} \approx \underset{m \times k}{Q} \underset{k \times n}{R},$$

where $R = \bar{R} \begin{pmatrix} I_k & \widehat{R}_{1:k}^{-1} \widehat{R}_{k+1:n} \end{pmatrix}$.

In practice, oversampling the matrix improves the robustness of the algorithm, and hence, the dimension of the sampled matrix B is given by $\ell = k + p$, where p is a small constant known as an oversampling parameter. In addition, the dimension of B is often much smaller than that of A (i.e., $\ell \ll m$, e.g., $\ell = 64$ and $m = 50,000$ in our experiments), and the cost of the deterministic QRCP factorization of B is marginal to the total cost. Hence, the overall cost of the algorithm is typically dominated by the first step of computing the sampled matrix B, which can be computed using communication-optimal kernels that exhibit high data locality and parallelism.

When the singular values of the matrix A decay slowly, the sampled matrix generated by the above algorithm may contain a significant amount of noise. To reduce the amount of noise, q iterations of the power method may be applied:

$$B = \Omega\, A\, (A^T A)^q.$$

This yields the following error bound on the approximation,

$$\|AP - QR\| \leq c(p, \Omega)^{1/(2q+1)}\, \sigma_{k+1},$$

where σ_{k+1} denotes the $(k+1)$-th largest singular value of A, and $c(p, \Omega)$ is a constant that depends on the oversampling parameter p and the sampling matrix Ω [8]. Since the condition number of B increases exponentially with q, to maintain the numerical stability in practice, the sampled matrix is orthogonalized after each application of A and A^T [15]. Figure 2 shows the pseudocode of the resulting algorithm.

In this paper, we focus on the fixed-rank problem to compute a rank-k approximation for a user-specified input parameter k. Alternatively, the fixed-accuracy problem seeks for a low-rank approximation whose approximation error is less than a user-specified tolerance ε. Figure 3 shows the pseudocode of our adaptive sample size scheme (adaptive-ℓ), which integrates the power iteration into the adaptive scheme for solving the fixed-accuracy problem by gradually increasing the size of the sampled subspace [8]. At each step of the adaptive-ℓ scheme, the sampled subspace is expanded by adding a new set of orthogonal basis vectors $B_{\ell+1:k}$ which are generated by the power iteration. To maintain the numerical stability, during each power iteration, after performing the matrix-matrix multiplication with A (or A^T), the new vectors $B_{j:k}$ (or $C_{j:k}$) are orthogonalized against the previous vectors $B_{1:j-1}$ (or $C_{1:j-1}$, e.g., using the Classical or Modified Gram Schmidt [7]), in addition to being orthogonalized against each other (e.g., using the Householder QR [7] or Cholesky QR [17]).

To reduce the cost of computing the approximation error, $\|A - AB_{1:\ell}^T B_{1:\ell}\|$, where $B_{1:\ell}$ stores the orthonormal basis vectors of the current sampling subspace, on Line 15, the adaptive-ℓ scheme estimates it by $\widetilde{\varepsilon} = \|\Omega(A - AB_{1:\ell}^T B_{1:\ell})\|$. This error estimate satisfies the following bound,

$$\|A - AB_{1:\ell}^T B_{1:\ell}\| \;\leq\; c_{ad} \sqrt{\frac{2}{\pi}}\, \widetilde{\varepsilon}, \tag{4}$$

with probability $1 - \min(m, n) c_{ad}^{-\ell_{inc}}$, where c_{ad} is a fixed constant [8]. Once the sampled matrix B is computed through the adaptive-ℓ scheme, the low-rank approximation can be computed by Steps 2 and 3 of random sampling.

Since the computed error $\widetilde{\varepsilon}$ is pessimistic, though the final approximation error is less than the user-specified ε, the adaptive scheme generally generates a sampled subspace whose dimension is greater than necessary. This induces the computational and storage overheads. In addition, compared to performing the matrix-matrix multiply with the final subspace all at once (e.g., fixed-rank problem), incrementally performing the matrix-matrix multiply with a smaller subspace at each step of the adaptive scheme often obtains lower performance. We study the performance of this adaptive scheme in Section 10.

4. IMPLEMENTATION

We now describe our GPU implementation of the random sampling algorithm of Figure 2. For Step 1 of the algorithm, we experimented with two types of sampling:

Require: Input: $m \times n$ matrix A.
1: Initialize:
$\ell := 0$, $Q := []$, and $\ell_{inc} := f(\ell, \ell_{init})$
e.g., $f(\ell, \ell_{inc}) = \ell_{inc}$ or $\ell_{init} + \ell$
2: $\Omega := \text{PRNG}(\ell_{inc}, m)$
3: $B := \Omega A$, and $C := []$
4: **repeat**
5: ▷ Expand sampled subspace
6: $k := \ell + \ell_{inc}$
7: $[B, C] := \text{POWER}(A, B, C, \ell + 1, k, q)$
8: $B_{\ell+1:k} := \text{QR}(B_{\ell+1:k})$
9: $\ell := k$
10: ▷ Generate new vectors
11: $\ell_{inc} := f(\ell, \ell_{inc})$
12: $\Omega := \text{PRNG}(\ell_{inc}, m)$
13: $B_{\ell+1:k} := \Omega A$, where $k := \ell + \ell_{inc}$
14: ▷ Compute approximation error
15: $\widetilde{\varepsilon} := \|B_{\ell+1:k} - B_{\ell+1:k} B_{1:\ell}^T B_{1:\ell}\|$
such that $\widetilde{\varepsilon} \approx \|A - A B_{1:\ell}^T B_{1:\ell}\|$
16: **until** $\widetilde{\varepsilon} \leq \varepsilon$
17: **return** $B := B_{1:\ell}$

Figure 3: Adaptive scheme to compute sampling subspace.

- Gaussian sampling: For Line 2 of Figure 2, we used the `cuRAND` library of NVIDIA to generate a Gaussian matrix Ω (matrix whose entries follow the standard normal distribution $\mathcal{N}(0, 1)$ with mean 0 and standard deviation 1). The sampling step then takes the form of a matrix-matrix multiply which is implemented using the general matrix-matrix multiply (GEMM) kernel from `cuBLAS` of NVIDIA.

- FFT sampling: we used the `cuFFT` library of NVIDIA to generate the sampled matrix B by applying an FFT transformation to A. Like many other FFT implementations, `cuFFT` obtains better performance for data sizes that are powers of two. Hence, in our experiments, we padded the matrix A with zeroes such that its leading dimension becomes the next power of two.

The sampling step $B = \Omega A$ consists of two steps, projection and sampling:

$$\underset{\ell \times n}{B} = \underset{\ell \times m}{S} \quad \underset{m \times m}{\Pi} \quad \underset{m \times n}{A}$$

where Π represents the projection matrix, S is the row selection matrix which randomly selects ℓ rows from Π or ΠA, and hence the sampling matrix Ω is given by $\Omega = S\Pi$. There are two sampling schemes, full and pruned sampling, which lead to different computational costs. In the full sampling scheme, the projected matrix ΠA is first computed, and then ℓ rows are selected, while in the pruned sampling scheme, the sampling matrix Ω is first computed, and then applied to A, or the ℓ rows are directly sampled from A (e.g., pruned FFT). Since only a small number of rows are selected through random sampling, compared to the full sampling scheme, the pruned sampling scheme may significantly reduce the computational cost. For Gaussian sampling, the random sampling matrix Ω is also Gaussian. Hence, we implement the pruned sampling scheme by first generating an ℓ-by-m Gaussian matrix Ω using `cuRAND`, and then computing the sampled matrix B through a matrix-matrix multiply (i.e., $B = \Omega A$). The flop count of this pruned sampling is $\mathcal{O}(mn\ell)$, while $\mathcal{O}(m^2 n)$ flops are needed for a full Gaussian sampling.

The pruned FFT only computes the ℓ selected rows and requires a fewer flops compared to the full FFT (i.e., $\mathcal{O}(mn \log(\ell))$ instead of $\mathcal{O}(mn \log(m))$). However, compared to the Gaussian sampling, the reduction in the flop count is less (i.e., a factor of $\mathcal{O}(\log(m)/\log(\ell))$ instead of $\mathcal{O}(m/\ell)$). In addition, since the execution time of FFT is not a function of only flop count (e.g., data access), the reduction in the execution time using the pruned FFT could be much less than $\mathcal{O}(\log(m)/\log(\ell))$, or the execution time could even increase. Though we provide a performance comparison of Gaussian and FFT sampling in Section 9, `cuFFT` does not support pruned FFT, and in this paper, we focus on Gaussian sampling, for which more theoretical work has been established [8].

Previously, the performance of several tall-skinny orthogonalization schemes on GPUs have been studied [19]. The results of the studies can be applied, on Lines 3 and 5 of the power iteration in Figure 2, to orthogonalize the $\ell \times n$ and $\ell \times m$ short-wide matrices B and C, respectively (i.e., $\ell < \min(m, n)$). In this paper, we focus on the Cholesky QR (CholQR) factorization [17] that obtains high performance based on BLAS-3 operations and can be implemented with minimum communication [5]. Specifically, CholQR computes the QR factorization of a matrix B in the following three steps[3]:

(i) Form a Gram matrix G; i.e., $G = BB^T$.

(ii) Compute the Cholesky factor $\bar{R}$ of the Gram matrix G; i.e., $\bar{R}^T \bar{R} := G$, where $\bar{R}$ is upper-triangular with non-negative diagonals.

(iii) Compute the orthogonal matrix Q by the backward-substitutions; i.e., $Q = \bar{R}^{-T} B$.

Similarly, on Line 10, CholQR is used to compute the QR factorization of the short-wide matrix C.

Classical and modified Gram Schmidt procedures (CGS and MGS, respectively) [7] are other well-known orthogonalization algorithms. While CGS orthogonalizes each column of the matrix against the previous columns one at a time, MGS orthogonalize each column against all the previous columns at once. Hence, MGS and CGS are based on BLAS-1 and BLAS-2 operations, respectively, and their performance is often lower than the BLAS-3 based CholQR. In addition, though the Householder QR (HHQR) [7] is an unconditionally accurate orthogonalization scheme, its performance is limited by the BLAS-1 and BLAS-2 operations, which obtain only a fraction of the GPU peak performance. In Section 8, we study the performance of these orthogonalization schemes. Though CholQR was stable in our experiments, it can be unstable for an ill-conditioned matrix A or for other choices of the parameters (e.g., k and p). This numerical issue may be overcome by reorthogonalizing the matrices, using HHQR for orthogonalizing B or when CholQR fails, using the Communication-Avoiding HHQR [5, 2], or

[3]This is the adaptation of CholQR to compute the LQ factorization of the short-wide matrix B.

Figure 4: Illustration of CholQR on two GPUs, where the dashed lines show the matrix distribution.

using mixed-precision arithmetic in CholQR [20]. Orthogonalization procedures for a stable and efficient implementation of the random sampling algorithm are part of our current research focus.

At Step 2, QRCP of the sampled matrix B is computed using the truncated QP3 on a GPU (on Line 6). Then, the tall-skinny QR factorization of the matrix $AP_{1:k}$ is computed using CholQR on the GPU (on Line 8). Finally, the upper-triangular matrix R is generated by the triangular solve and triangular matrix-multiply on the GPU (Lines 9 and 10). Since the dimension of the sampled subspace is much smaller than that of A, the computational and communication costs at Steps 2 and 3 are of lower order than that of the first step to generate the sampled subspace. We provide the computation and communication costs of each step in Section 5.

Finally, to utilize multiple GPUs, the matrix A is distributed in a 1D block row format among the GPUs such that each GPU owns about the same number of rows (i.e., the i-th GPU owns the block row $A_{(i)}$ of size c-by-n, where $c \approx m/n_g$ and n_g is the number of available GPUs). Both matrices Ω and C are distributed in the same 1D block column format as that of A^T. Then, on Line 3 of random sampling or on Line 12 of the power iteration, the i-th GPU computes the partial result $B_{(i)}$ of the sampled matrix B by performing the local matrix-matrix multiplication of $\Omega_{(i)}$ or $C_{(i)}$ with $A_{(i)}$, respectively (e.g., $B_{(i)} := C_{(i)}A_{(i)}$). Next, the CPU accumulates the partial results $B_{(i)}$ to form the $\ell \times n$ sampled matrix B (i.e., $B := \sum_i^{n_g} B_{(i)}$). Since the dimension of B is small (i.e., $\ell < n \ll m$), we compute the QR factorization of B on the CPU (using either CholQR or HHQR), and the resulting orthogonal matrix is copied to the GPU such that it is duplicated on each GPU. Finally, each GPU performs its local matrix-matrix multiply to compute the sampled matrix C which is distributed in the same 1D block column format as that of A^T (i.e., $C_{(i)} := BA_{(i)}^T$).

Now, to perform CholQR of C on multiple GPUs, each GPU first computes the local matrix-matrix multiply, $G_{(i)} := C_{(i)}C_{(i)}^T$, and then sends the result to the CPU, where the Gram matrix $G := \sum_{i=1}^{n_g} G_{(i)}$ is computed. The Cholesky factor $\bar{R}$ of the matrix G is then computed on CPU. Finally, the CPU broadcasts the Cholesky factor $\bar{R}$ to all the GPUs, and each GPU independently performs the substitution, $\bar{Q}_{(i)} := \bar{R}^{-T}C_{(i)}$ (i.e., $\bar{Q}\bar{R} = C$). Figure 4 illustrates our multi-GPU CholQR implementation.

Finally, for Steps 2 and 3 of random sampling, the truncated QP3 of the sampled matrix B, and the triangular solve and multiply to compute R are performed on a GPU (on Lines 6, 9, and 10), while the QR factorization of $AP_{1:k}$ is computed based on the multi-GPU CholQR (on Line 8).

5. PERFORMANCE MODEL

Figure 5 compares the computational and communication costs of our random sampling implementation on one GPU with those of QP3 and its communication-avoiding variant [4]. Here, the computational and communication costs are measured, respectively, based on the flop count and the words transferred between the two levels of the local memory hierarchy, where the the size of the fast memory is M.

	#flops	#words
Random sampling		
Sampling (Gaussian)	$\mathcal{O}(mn\ell)$	$\mathcal{O}(mn\ell/M^{1/2})$
Sampling (FFT)	$\mathcal{O}(mn\log(m))$	$\mathcal{O}(mn\log(m)/\log(M))$
Iter. (mult.)	$\mathcal{O}(mn\ell q)$	$\mathcal{O}(mn\ell q/M^{1/2})$
Iter. (orth.)	$\mathcal{O}((m+n)\ell^2 q)$	$\mathcal{O}((m+n)\ell^2 q/M^{1/2})$
QRCP	$\mathcal{O}(n\ell^2)$	$\mathcal{O}(n\ell^2)$
QR	$\mathcal{O}(m\ell^2)$	$\mathcal{O}(m\ell^2/M^{1/2})$
Total	$\mathcal{O}(mn\ell(1+2q))$	$\mathcal{O}(mn\ell(1+2q)/M^{1/2})$
QP3	$\mathcal{O}(mnk)$	$\mathcal{O}(mnk)$
CAQP3	$\mathcal{O}(mn(m+n))$	$\mathcal{O}(mn^2/M^{1/2})$

Figure 5: Computation and communication costs on one GPU.

- Sampling (Step 1): The sampling is either based on an FFT or a matrix-matrix multiply, and both can use a communication-optimal kernel [10].
- Power iteration (Step 1): each iteration performs two matrix-matrix multiplies and orthogonalization of two matrices, one with the dimension $\ell \times m$ and the other with the dimension $\ell \times n$. Communication-optimal orthogonalization procedures [5, 17] exist, which can be used for this step.
- QRCP (Step 2): since the sampled subspace is small compared to the global space (i.e., $\ell \ll m$), this step only has a marginal computational and communication costs. Our implementation is based on the standard QP3 algorithm, but a communication-optimal variant of QP3 [4] can be used for this step.
- QR (Step 3): just as for the power iteration, a communication-optimal orthogonalization procedure can be used for this step.

Thus, both the computation and communication costs on one GPU are dominated by the matrix-multiply kernel. The performance model can be extended to multiple GPUs, where the matrix-multiply kernel remains the bottleneck with #flops $= \mathcal{O}(\frac{mn\ell(1+2q)}{n_g})$ and #words $= \mathcal{O}(\frac{mn\ell(1+2q)}{n_g M^{1/2}})$ [9].

6. EXPERIMENTAL SETUPS

In the following four sections, we studied the accuracy and performance of random sampling using three different matrices A. The first two matrices A were generated by $A := X\Sigma Y$ with randomly generated orthogonal matrices X and Y, and a diagonal matrix Σ shown in Table 1. The last matrix comes from the International Hapmap Project [1]. We used the latest bulk release (as of August 1, 2014). Each row of A corresponds to a specific nucleotide basis and a column corresponds to an individual from a specific population. We extracted the data using the first five chromosomes and from four different populations: Utah residents with Northern and Western European ancestry, Gujarati Indians

	Matrix Name		
	POWER	EXPONENT	HAPMAP
σ_i	$(i+1)^{-3}$	$10^{-i/10}$	$--$
σ_0	1	1	9.9e+03
σ_{k+1}	8e-06	1.3e-05	5e+02
$\kappa(A)$	1.3e+05	7.9e+04	2e+01
m	500,000	500,000	503,783
n	500	500	506
k	50	50	50
p	10	10	10
ℓ	60	60	60

Table 1: Test matrices.

	QP3	$q=0$	$q=1$	$q=2$
POWER	4.47e-05	9.08e-05	4.59e-05	4.45e-05
EXPONENT	2.69e-05	5.18e-05	2.69e-05	2.69e-05
HAPMAP	5.99e-01	9.86e-01	8.74e-01	8.18e-01

Figure 6: Approximation error norm $\|AP-QR\|/\|A\|$.

in Houston, Texas, Japanese in Tokyo, Japan, and Yoruban in Ibadan, Nigeria. Computing a low-rank approximation on such data can be used for population clustering [6, 13].

In Section 7, we first compare the approximation errors of QP3 and random sampling using the fixed parameters shown in Table 1. Then, in Section 8, we study the performance of the GPU kernels, and in Section 9, we study the performance of random sampling over ranges of parameters (i.e., $m = 2,500 \sim 50,000$, $n = 500 \sim 5,000$, $\ell = 32 \sim 512$, and $q = 0 \sim 12$). Finally in Section 10, we discuss the performance of the adaptive scheme for solving the fixed-accuracy problem. To the best of our knowledge, this is the first experimental studies of the adaptive scheme. All the experiments were conducted in the 64-bit double precision.

Since the condition numbers of the sampled matrices B and C increase exponentially with q, the approximation error diverged without orthogonalization. To avoid the numerical issue, in our experiments, we orthogonalized both sampled matrices using CholQR with one full reorthogonalization, which made the sampling algorithm stable. All the codes were compiled using the C++ GNU compiler `gcc` (version 4.4.7) and the NVIDIA compiler `nvcc` (CUDA version 6.0.1), with the optimization flag `-O3`, and linked to threaded MKL (version 10.3). We conducted our experiments on two eight-core Genuine Intel(R) 2.60GHz CPUs and three NVIDIA Tesla K40c GPUs.

7. NUMERICAL RESULTS

In Figure 6, we compare the approximation errors of the deterministic QP3 and the random sampling using the fixed values of the parameters shown in Table 1. Though the approximation error decreased with the number of iterations, random sampling without power iteration (i.e., $q = 0$) obtained the approximation with the same order of error as QP3. These results were obtained for an oversampling equal to $p = 10$. Without oversampling (i.e., $p = 0$), the error norm was about an order of magnitude greater. In addition, a greater oversampling (e.g., $p = 20$ or 50) could further improve the accuracy, but with a smaller factor (i.e., the constant factor $C(\Omega, p)$ is roughly proportional to $p^{-1/2}$ [8]).

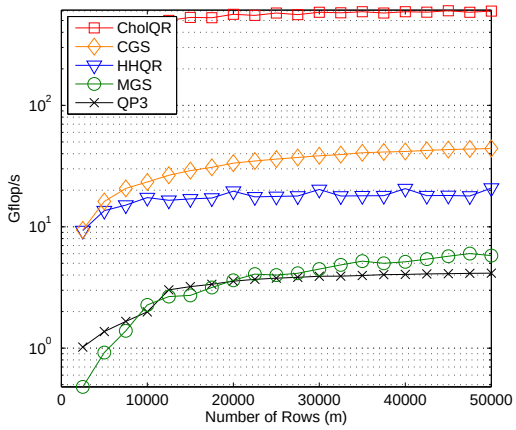


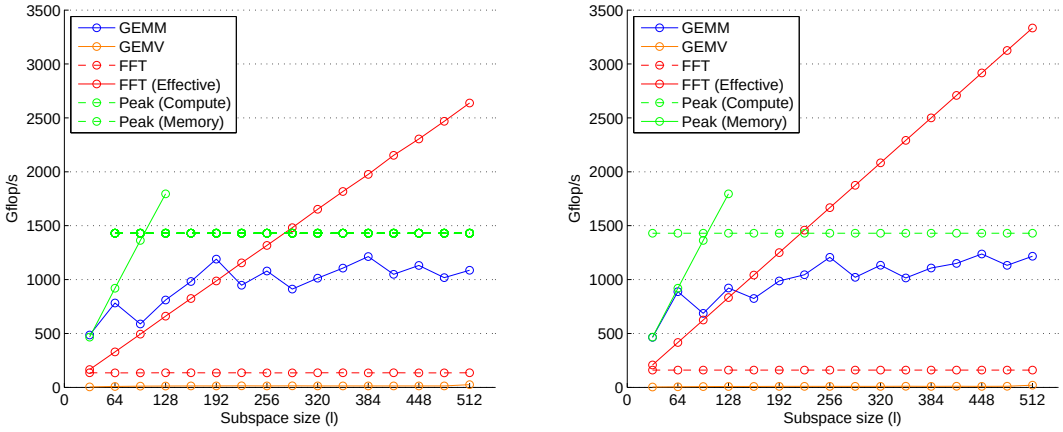
Figure 7: Performance of QP3 and tall-skinny QR.

These numbers are reported for Gaussian sampling, but FFT sampling gave the approximation errors of the same order.

8. KERNEL PERFORMANCE

Before studying the performance of random sampling in the next section, in this section, we study the performance of our GPU kernels which are used for random sampling. First, to study the cost of the QP3 factorization, in Figure 7, we compare the QP3 performance with the performance of other orthogonalization algorithms on the GPU, i.e., Householder QR (HHQR), Cholesy QR (CholQR), and the classical and modified Gram Schmidt (CGS and MGS). For our performance studies, we focused on the tall-skinny matrices (i.e., $m \gg n$), and varied the number of rows while fixing the number of columns in A (i.e., $m = 2,500 \sim 50,000$ while $n = 64$). In the figure, we see that HHQR was about $5\times$ faster than QP3, indicating the cost of column pivoting. In addition, BLAS-3 based CholQR obtained the speedups of up to 33.2 and an average speedup of 30.5 over HHQR, demonstrating the cost of the intra GPU communication associated with the BLAS-1 and BLAS-2 operations required by HHQR. The figure also shows that due to the intra GPU communication, HHQR, which uses both BLAS-1 and BLAS-2, was faster than MGS but slower than CGS because our MGS and CGS perform most of their flops using BLAS-1 and BLAS-2, respectively.

Next, in Figure 8, we compare the performance of the full FFT sampling with the performance of the matrix-matrix multiply (GEMM) used for the pruned Gaussian sampling. For the row sampling in Figure 8(a) (i.e., $B = \Omega\ A$), we varied the dimension of the sampled matrix B from $\ell = 32$ to 512 for a fixed $50,000 \times 2,500$ input matrix A (i.e., sampling about 0.06 to 1.02% of the rows of A). In the figure, we also show the peak performance for the double-precision flop (i.e., 1,430 Gflop/s) and the peak performance based on the memory bandwidth (i.e., 288 GB/s, assuming blocksize of 512). The matrix-matrix multiply used for the pruned Gaussian sampling exhibits a regular memory access pattern and a high level of data parallelism. Hence, it can be optimized for the intra GPU communication and obtain a near peak performance (i.e., about 1,200 Gflop/s). As a result, since the multiplication requires $\mathcal{O}(mn\ell)$ flops, for



(a) Row sampling.

(b) Column sampling.

Figure 8: Performance of pruned Gaussian and full FFT sampling.

a large enough sampling size, the sampling time increases linearly with the sampling size ℓ. On the other hand, the full FFT sampling performs only $\mathcal{O}(mn\log(m))$ flops (e.g., $\log(m) \approx 15.6$ when $m = 50,000$). As a result, compared to the matrix-matrix multiply, though its performance is often lower (e.g., about 135 Gflop/s in our experiments), the full FFT sampling can be as fast as the pruned Gaussian sampling, and it was faster when $\ell > 192$. This can be seen in the figure as the Gflop/s of the pruned Gaussian sampling becomes greater than the "effective" Gflop/s of the full FFT sampling, which is computed as the ratio of the number of flops required for the pruned Gaussian sampling over the full FFT sampling time. The figure also shows that the matrix-vector multiply (GEMV), which is used to implement CGS, HHQR, and QP3, obtains much lower performance than the matrix-matrix multiply.

Similarly, Figure 8(b) compares the performance of the full FFT column sampling with the pruned Gaussian column sampling (i.e., $B = \Omega\ A^T$). Again, the Gaussian sampling obtained the near peak performance, but the full FFT was faster when $\ell > 128$. For the rest of the paper, we focus on the pruned Gaussian (row) sampling with a small sampling size since more theoretical work has been established for the Gaussian sampling [8].

Besides sampling, computing the orthogonal basis vectors of the sampled subspace during the power iteration can become expensive. Specifically, on Lines 5 and 10 of POWER(A, B, C, j, k, q) in Figure 2, we compute the QR factorization of the short-wide matrices B and C, respectively. While Figure 7 shows the performance of CholQR for tall-skinny matrices, Figure 9 shows the performance for the short-wide matrices with the same number of rows but with different numbers of columns (i.e., $m = 64$ and $n = 2,500 \sim 50,000$). Again, CholQR showed excellent performance, obtaining speedups of up to 106.4 and the average speedup of 72.9 over HHQR.

Since the execution time of the random sampling is dominated by the sampling and orthogonalization phases, we can estimate the performance based on the performance results in Figures 7 through 9. This allows us to evaluate the performance of random sampling on a target computer before implementing the algorithm or to verify the performance of the existing implementation. For instance, Figure 10 shows the estimated performance of the random sampling and that of the truncated QP3 for $m = 2,500 \sim 50,000$ with $n = 2,500$ and $(\ell; p) = (64; 10)$. We see that due to its communication costs, QP3 could not fully utilize the computational power of the GPU, and its performance was limited under 29 Gflop/s. On the other hand, random sampling is expected to better utilize the hardware, reaching 676 Gflop/s for $q = 1$ and 489 Gflop/s for $q = 0$. Hence, random sampling is expected to obtain 23.8 or 17.1 times higher Gflop/s than QP3 when $q = 1$ or 0, respectively. In addition, since random sampling with $q = 1$ performs roughly 3.6 times more flops than QP3, we expect the random sampling to obtain the speedup of $23.8/3.6 = 6.7$ over QP3.

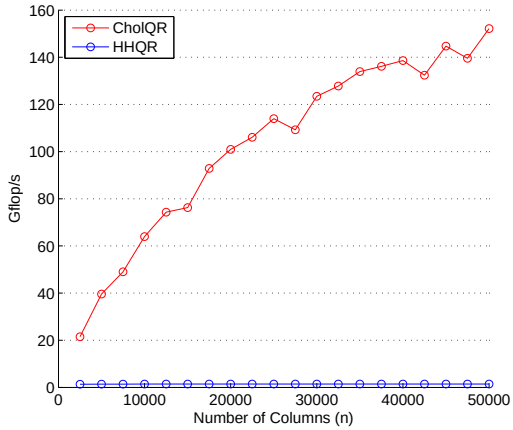


Figure 9: Performance of short-wide QR.

9. PERFORMANCE RESULTS

We now study the performance of random sampling on two eight-core Intel SandyBridge CPUs with an NVIDIA K40c

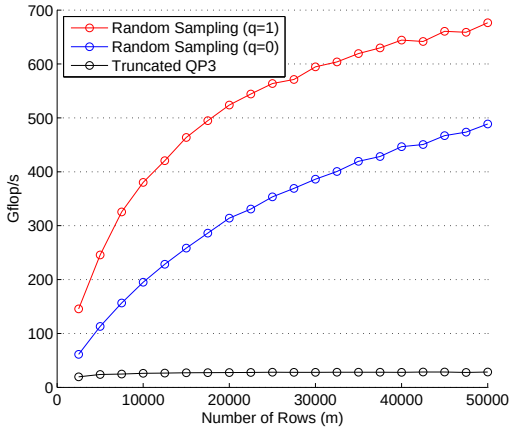


Figure 10: Estimated Gflop/s of random sampling and QP3.

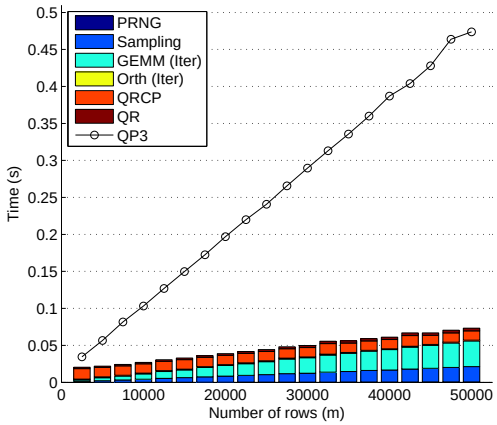


Figure 11: Random sampling and QP3 time with different numbers of rows, where "Sampling" and "GEMM (iter)" in the legends corresponds to the matrix-matrix multiply with the initial sampling matrix Ω and the matrix-matrix multiplies during the power iterations, respectively.

GPU. First, Figure 11 shows the total execution time of random sampling and QP3 with the same number of columns but with different numbers of rows in A (i.e., $n = 2,500$ and $m = 2,500 \sim 50,000$ with $(k; p; q) = (54; 10; 1)$). Both computational and communication costs of both random sampling and QP3 depend linearly on the number of rows, m (see Figure 5), and their execution time also increased linearly with m. However, the QP3 factorization time increased at a faster rate (i.e., QP3 time $\approx 9.34m10^{-6} + 0.0098$, while random sampling time $\approx 1.15m10^{-6} + 0.0162$). As a result, random sampling obtained speedups of up to 6.6 and the average speedup of 5.1 over QP3. For these experiments, we performed one power iteration (i.e., $q = 1$). We saw in Figure 6 that even without power iteration (i.e., $q = 0$), the approximation error norm of random sampling was already in the same order of magnitude as that of QP3. Without power iteration, the random sampling obtained speedups of up to 12.8 and the average speedup of 8.8. The speedup of 6.6 obtained for $q = 1$ agrees with our estimate in Figure 10.

Figure 11 also shows that for a small m, the QRCP step remained the bottleneck. However, for a large enough m, the overall run time of random sampling was dominated by the first step of computing the sampled matrix B. For example, when $m = 50,000$, about 78% of the total run time was spent in the first step, which includes the generation of the sampling matrix Ω, the sampling time, the matrix-matrix multiply in the power iteration, and the orthogonalization (0.9, 28.3, 47.3, and 1.4% of the overall time, respectively). The run time of random sampling was thus dominated by the matrix-matrix multiply (i.e., about 75% of the overall time). This is one of the main attractive properties of random sampling since this BLAS-3 operation can be tuned to exploit high data locality and parallelism, while QP3 performs a half of its total flops using BLAS-2 that obtains much lower performance (around 30 Gflop/s).

Figure 12 shows the QP3 and random sampling time with different numbers of columns in A (i.e., $n = 500 \sim 5,000$ with $m = 50,000$ and $(\ell; p; q) = (64; 10; 1)$). Again, compared to the random sampling, the QP3 time increased much quicker with the increase in the number of columns (i.e., QP3 time $\approx 1.80n10^{-4} + 181.77$, while random sampling time $\approx 0.2119n10^{-2} + 239.7$). Similarly, Figure 13 shows the execution time with varying target rank k (i.e., $\ell = 32 \sim 512$ with $(m; n) = (50,000; 2,500)$ and $(p; q) = (10; 1)$). The QP3 time also increased quicker with the increase in the target rank (i.e., QP3 time $\approx 0.81\ell10^{-2} - 0.0235$, while random sampling time $\approx 0.10\ell10^{-2} + 0.0227$). At the end, random sampling outperformed QP3 over large ranges of parameters.

Figure 14 compares the QP3 run time with that of the random sampling with different numbers of power iterations (i.e., $q = 0 \sim 12$). As expected, we see that the run time of random sampling increases linearly with q, and that random sampling outperforms QP3 for up to twelve iterations (i.e., $q \leq 12$). We note that for our test matrices, the random sampling without iteration (i.e., $q = 0$) computed an approximation whose error norm is in the same order of magnitude as QP3.

Finally, Figure 15 shows the parallel strong scaling of random sampling over three Kepler GPUs, using the fixed parameters $(m; n) = (150,000; 2,500)$ and $(\ell; p; q) = (64; 10; 1)$. On two and three GPUs, the respective parallel speedups of the matrix-matrix multiply were about 2.8 and 5.1. These superlinear speedups are due to the fact the chunks $A_{(i)}$ on each GPU get less tall and skinny w hen the number of GPUs n_g grows, and e found that the efficiency of the GPU GEMM kernel increases as the matrix becomes closer to square: it is around 440, 630 and 760 Gflop/s with 1, 2, and 3 GPUs (i.e., $m/n_g = 150,000$, 75,000, and 50,000), respectively. With the communication optimal CholQR, inter-GPU communications only represented 1.6% of total time for two GPUs, and 4.3% for three GPUs. In the end, random sampling obtained an overall speedup of about 2.4 and 3.8 on two and three GPUs, respectively.

10. ADAPTIVE PERFORMANCE

Figure 16 shows the convergence of the error estimate $\widetilde{\varepsilon}$ computed at each step of the adaptive-ℓ scheme. For this experiment, we used the $50,000 \times 2,500$ EXPONENT matrix, and computed its low-rank approximations without power

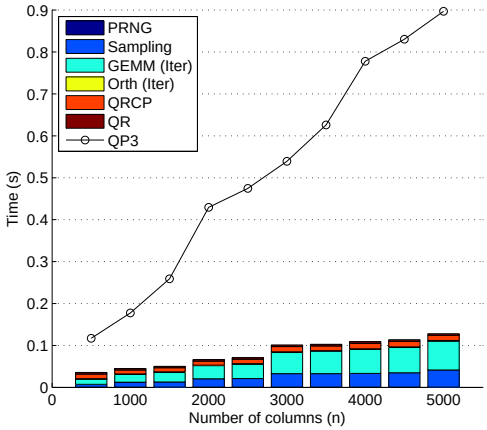


Figure 12: Random sampling and QP3 time with different numbers of columns.

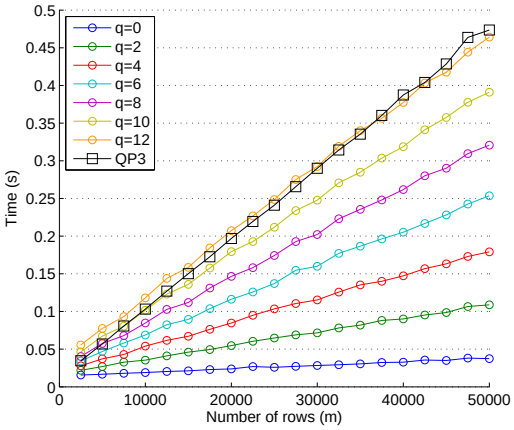


Figure 14: Random sampling and QP3 time with different numbers of iterations.

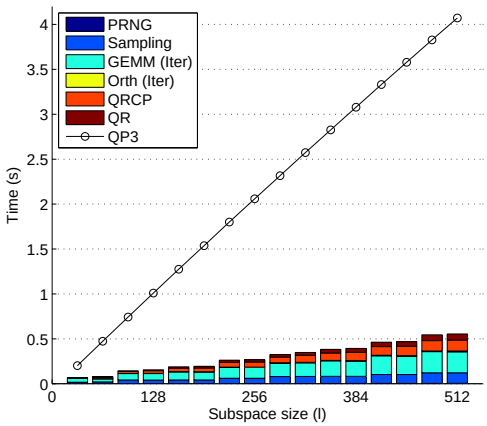


Figure 13: Random sampling and QP3 time with different target ranks.

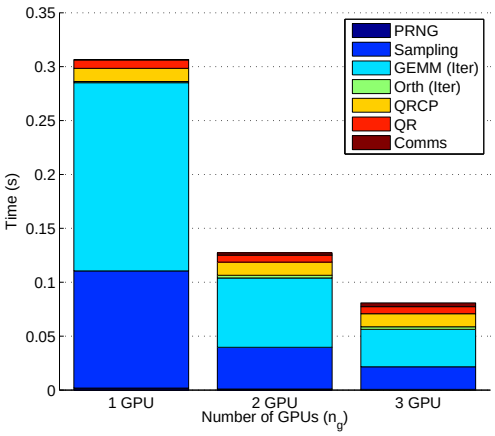


Figure 15: Strong parallel scaling over 3 GPUs.

iteration (i.e., $q = 0$). Each line corresponds to a different static parameter ℓ_{inc}, the amount by which the subspace size is increased at each iteration (i.e., $f(\ell, \ell_{inc}) = \ell_{inc}$). We started with the same initial subspace size (i.e., $\ell_{init} = 8$), and iterated until the error estimate $\widetilde{\varepsilon}$ was smaller than 10^{-12} (i.e., $\epsilon = 10^{-12}$).

In the figure, the dashed black line shows the actual errors $\|A - AQ^TQ\|_2$, which were one or two order of magnitude less than the error estimates $\widetilde{\varepsilon}$, which are the probabilistic estimates, satisfying (4). For example, with a fixed probability of failure, γ, the constant c_{ad} in (4) is given by $c_{ad} = (\gamma/\min(m,n))^{-1/\ell_{inc}}$, where $(\gamma/\min(m,n)) < 1$. Thus, a larger value of the parameter ℓ_{inc} decreases the constant c_{ad}, making the error estimate $\widetilde{\varepsilon}$ less pessimistic. This can be observed in Figure 16, where the error estimates $\widetilde{\varepsilon}$ with $\ell_{inc} = 8$ were slightly larger and worse than the estimates with a larger value of ℓ_{inc}. In addition, a larger value of static ℓ_{inc} has a greater chance of overestimating the sampling size which would satisfy the tolerance ϵ. This increases the computation and storage costs of the random sampling.

Figure 17 shows the same convergence of the error estimate $\widetilde{\varepsilon}$ but now with respect to the elapsed time in second. We see that the convergence is slower using a smaller value of ℓ_{inc}. This is because the performance of the GPU kernel degrades for a smaller dimension of the input matrices (see Figure 18). Hence, there is a trade-off when selecting the static parameter ℓ_{inc}: a larger ℓ_{inc} improves the efficiency of the GPU kernels, but it increases the chance of overestimating the size of the required sampling subspace. One potential solution is to adjust the parameter ℓ_{inc} based on the convergence of the error estimates. For example, we show the result of simple linear interpolation of the previous two steps to select the next ℓ_{inc}. It works well for this particular matrix, but we are working on other adaptive schemes based on the performance and numerical measurements gathered over the previous adaptive steps and power iterations.

11. CONCLUSION

In this paper, we compared the performance of a deterministic QRCP with that of a random sampling algorithm on a GPU. While QRCP requires synchronization and communication at each step of the factorization, random sampling can be implemented using communication-optimal kernels.

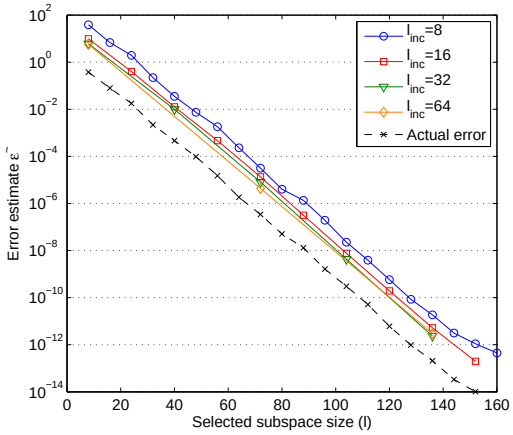


Figure 16: Convergence of estimated error with adaptive sampling size.

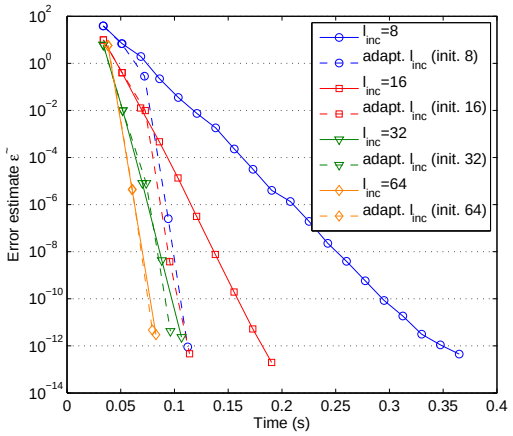


Figure 17: Convergence of estimated error in term of time, with adaptive sampling dimension using static or adaptive parameter ℓ_{inc}.

Our performance results on an NVIDIA Kepler GPU demonstrated that the random sampling can obtain a speedup of up to 12.8 over QRCP, while achieving a comparable approximation accuracy. We then studied the parallel scaling of the random sampling over multiple GPUs, and showed that the random sampling can obtain a nearly-linear speedup over three GPUs. Due to its communication efficiency, we expect the performance benefits of random sampling to increase on a computer with higher communication cost, like a distributed-memory computer. The GPU kernels developed for this studies will be released as a part of the `MAGMA` software package. Hence, our primary focus was to improve the performance and robustness of the algorithm in practice so that it can be used in many applications.

To improve the performance and stability of random sampling, we are studying other orthogonalization schemes including Communication-Avoiding QR [5] and mixed-precision CholQR [20], and an adaptive scheme based on the numerical properties of the matrices at run time. We plan to study the performance of our implementation for real applications and compare it with other algorithms including the communication-avoiding QP3 [4]. In particular, we will investigate other error measurements (e.g., clustering errors) to better understand the quality of the approximation computed by different algorithms.

	ℓ_{inc}				
	8	16	32	48	64
Gflop/s	123.3	247.0	489.5	597.8	778.5

Figure 18: Performance of GEMM used for adaptive scheme.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy Office of Science under Award Number DE-FG02-13ER26137/DE-SC0010042, the U.S. National Science Foundation under Award Number 1339822, and the Russian Scientific Fund, Agreement N14-11-00190.

12. REFERENCES

[1] The international HapMap project. Nature, 426(6968):789–796, 12 2003.

[2] M. Anderson, G. Ballard, J. W. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In Proceedings of the IEEE International Parallel Distributed Processing Symposium, pages 48–58, May 2011.

[3] P. Businger and G. Golub. Linear least squares solutions by Householder transformations. Numerische Mathematik, 7:269–276, 1965.

[4] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication avoiding rank revealing QR factorization with column pivoting. Technical Report UCB/EECS-2013-46, University of California, Berkeley, 2013.

[5] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. SIAM Journal on Scientific Computing, 34(1):A206–A239, 2012.

[6] P. Drineas, M. Mahoney, and S. Muthukrishnan. Relative-error CUR matrix decompositions. SIAM Journal on Matrix Analysis and Applications, 30(2):844–881, 2008.

[7] G. Golub and C. van Loan. Matrix Computations. The Johns Hopkins University Press, Baltimore, MD, 4th edition, 2012.

[8] N. Halko, P.-G. Martinsson, and J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM Review, 53(2):217–288, 2011.

[9] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. Journal of Parallel and Distributed Computing, 64(9):1017–1026, 9 2004.

[10] H. Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[11] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert. Randomized algorithms for the low-rank approximation of matrices. Proceedings of the National Academy of Sciences, 104(51):20167–20172, 2007.

[12] M. W. Mahoney. Randomized algorithms for matrices and data. Found. Trends Mach. Learn., 3(2):123–224, 2011.

[13] M. W. Mahoney and P. Drineas. CUR matrix decompositions for improved data analysis. Proceedings of the National Academy of Sciences, 106(3):697–702, 2009.

[14] P.-G. Martinsson, V. Rokhlin, and M. Tygert. A randomized algorithm for the decomposition of matrices. Applied and Computational Harmonic Analysis, 30(1):47–68, 1 2011.

[15] P.-G. Martinsson, A. Szlam, and M. Tygert. Normalized power iterations for the computation of SVD. Technical report, 2010.

[16] G. Quintana-Ortí, X. Sun, and C. H. Bischof. A BLAS-3 version of the QR factorization with column pivoting. SIAM Journal on Scientific Computing, 19(5):1486–1494, 1998.

[17] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. SIAM J. Sci. Comput., 23:2165–2182, 2002.

[18] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert. A fast randomized algorithm for the approximation of matrices. Applied and Computational Harmonic Analysis, 25(3):335–366, 11 2008.

[19] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. Improving the performance of CA-GMRES on multicores with multiple GPUs. In Proceedings of the IEEE International Parallel and Distributed Symposium, pages 382–391, 2014.

[20] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra. Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs. In International meeting on high-performance computing for computational science, 2014.