SIAM J. SCI. COMPUT.
Vol. 41, No. 4, pp. C339–C366

RANDOMIZED GPU ALGORITHMS FOR THE CONSTRUCTION OF HIERARCHICAL MATRICES FROM MATRIX-VECTOR OPERATIONS[*]

WAJIH BOUKARAM[†], GEORGE TURKIYYAH[‡], AND DAVID KEYES[†]

Abstract. Randomized algorithms for the generation of low rank approximations of large dense matrices have become popular methods in scientific computing and machine learning. In this paper, we extend the scope of these methods and present batched GPU randomized algorithms for the efficient generation of low rank representations of large sets of small dense matrices, as well as their generalization to the construction of hierarchically low rank symmetric $\mathcal{H}^2$ matrices with general partitioning structures. In both cases, the algorithms need to access the matrices only through matrix-vector multiplication operations which can be done in blocks to increase the arithmetic intensity and substantially boost the resulting performance. The batched GPU kernels are adaptive, allow nonuniform sizes in the matrices of the batch, and are more effective than SVD factorizations on matrices with fast decaying spectra. The hierarchical matrix generation consists of two phases, interleaved at every level of the matrix hierarchy. A first phase adaptively generates low rank approximations of matrix blocks through randomized matrix-vector sampling. A second phase accumulates and compresses these blocks into a hierarchical matrix that is incrementally constructed. The accumulation expresses the low rank blocks of a given level as a set of local low rank updates that are performed simultaneously on the whole matrix allowing high-performance batched kernels to be used in the compression operations. When the ranks of the blocks generated in the first phase are too large to be processed in a single operation, the low rank updates can be split into smaller-sized updates and applied in sequence. Assuming representative rank k, the resulting matrix has optimal $O(kN)$ asymptotic storage complexity because of the nested bases it uses. The ability to generate an $\mathcal{H}^2$ matrix from matrix-vector products allows us to support a general randomized matrix-matrix multiplication operation, an important kernel in hierarchical matrix computations. Numerical experiments demonstrate the high performance of the algorithms and their effectiveness in generating hierarchical matrices to a desired target accuracy.

Key words. hierarchical matrices, GPU, randomized algorithms, low rank factorization, low rank updates, batched algorithms, matrix-matrix multiplication, matrix compression, nested bases

AMS subject classifications. 15A23, 65F30, 68N19, 68W10, 68W20, 68W25

DOI. 10.1137/18M1210101

1. Introduction.

Randomized methods for computing low rank approximations and related factorizations of large dense matrices have become quite popular in recent years [17]. Randomized factorization has proven to be robust, accurate, and able to exploit modern computing architectures including, in particular, high performing arithmetically intensive BLAS3 GEMM kernels. These methods construct low dimensional subspaces that approximate the range of the matrix by randomized matrix-vector sampling and then restrict the matrix to this subspace to compute desired low rank factorizations. The accuracy of the subspace approximation can be adaptively controlled and oversampling in the generation of the range approximation guarantees

[*]Submitted to the journal's Software and High-Performance Computing section August 28, 2018; accepted for publication (in revised form) May 9, 2019; published electronically July 9, 2019.
https://doi.org/10.1137/18M1210101

Funding: This work was supported by KAUST through the Extreme Computing Research Center and project OSR-2018-CARF-3666. The work of the second author was supported by QNRF under grant NPRP-5-995-2-415.

[†]Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal 23955-6900, Saudi Arabia (wajihhalim.boukaram@kaust.edu.sa, david.keyes@kaust.edu.sa).

[‡]Department of Computer Science, American University of Beirut, Beirut, Lebanon 1107-2020 (gt02@aub.edu.lb).

a negligible risk for the randomized methods in failing to meet target accuracy above machine precision.

In this paper, we seek to expand the scope of these powerful methods in two directions. The first direction is in the context where low rank factorizations are desired for a large batch of relatively small matrices instead of a single large matrix. These batched factorizations are essential building blocks in many modern linear algebra algorithms [30]. Machine learning libraries, particularly those targeting GPUs such as TensorFlow and Torch [1, 10], rely on batched factorizations for obtaining performance and there is a need to expand batched linear algebra routines to include high performing low rank factorization methods. While one could rely on recently developed Jacobi batched SVD kernels [7] and truncate the SVD factorizations for generating low rank approximations, adaptive randomized methods can be more efficient in the case when the matrices of the batch are known to have fast decaying spectra.

The second direction concerns the case of a large dense matrix that does not admit a globally low rank factorization but various off-diagonal blocks of it, at different levels of granularity, do admit such low rank approximations resulting in a hierarchically low rank representation of the matrix. These hierarchical matrices, also known as $\mathcal{H}$-matrices [16], have proven to be a very powerful and practical representation for the large matrices that arise in many applications involving the discretization of integral equations, covariance matrices in statistics, Schur complements, preconditioners for elliptic equations, Hessians in optimization, and many others. In some applications, the hierarchical matrix approximation may be generated directly using analytically available kernels. But in many applications of interest, these analytical construction techniques are not applicable and algebraic methods are needed. We therefore seek to adapt randomized methods to the generation of hierarchically low rank approximations of these large dense matrices. Our primary interest is to develop the construction algorithm when the access to the matrix is available through matrix-vector multiplication operations only. The full dense form of the hierarchical matrix is far too expensive to generate and store, and therefore it is impractical to start from such a representation and then compress it. The methods we develop here directly construct the compressed hierarchical matrix form of the matrix without directly accessing its entries.

The hierarchical matrix representation we target here is the $\mathcal{H}^2$ representation with a general partitioning structure. General partitioning structures allow low rank blocks to appear anywhere and be of any size in the matrix. Because of their flexibility, general partitioning structures, also referred to as standard admissibility [3], are able to adapt their structure and the location of the dense blocks to the problem at hand. This produces lower numerical ranks in the low rank matrix blocks than what would be obtained using a partitioning with a fixed pattern and results in lower storage requirements for a given accuracy. A commonly used fixed-pattern partitioning structure is the weak admissibility structure (see Figure 1) which, while simpler and easier to parallelize and manipulate, constrains every off-diagonal block to be represented as single low rank block, often resulting in large numerical ranks that may affect performance and storage requirements. General partitioning schemes (middle and right in Figure 1) can place dense blocks anywhere in the matrix where short range interactions need to be represented accurately and can result in memory savings particularly when moderate accuracy is needed. The $\mathcal{H}^2$ representation requires that the bases used in the low rank representations are nested, i.e., the basis of a block column or block row in the matrix, be obtained from the bases of children,

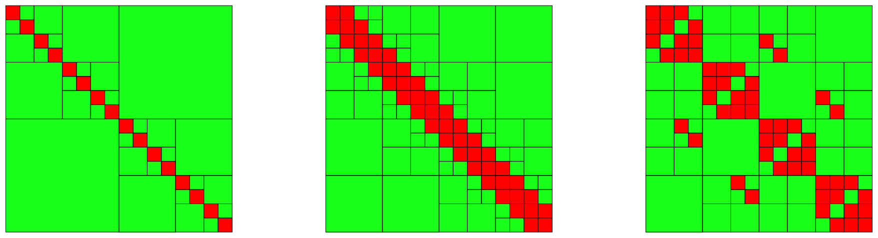


FIG. 1. *Weak versus standard admissibility partitioning of a hierarchical matrix. The simplicity of the weak admissibility structure (left) comes at the cost of suboptimal performance in both memory and flops. The general partitioning structures (middle and right) can represent matrices more efficiently memorywise and, when coupled with nested bases, have asymptotically optimal memory complexity.*

i.e., finer blocks. This allows the representation to achieve asymptotically optimal memory requirements [3], scaling linearly with the problem size.

GPUs are our primary target architecture. Their ubiquity in scientific workstations makes it important to develop algorithms and implementations that leverage their compute capabilities. While the algorithms we present may be used on CPUs, many of our design decisions are guided by the architecture of modern GPUs, particularly the small available memory. In fact, one of our motivations for targeting the $\mathcal{H}^2$ representation, despite the additional complexities introduced by using nested bases and general partitioning structures, is its asymptotically optimal memory requirements. Our construction algorithms also limit the memory needed during execution by recompressing periodically, as will be described in section 3. This allows a broader range of problems to take advantage of GPU-hosted algorithms.

The paper is organized as follows. The rest of this section describes prior work on the hierarchical matrix construction problem and defines some notation used in the presentation. Section 2 describes a batched adaptive randomized approximation (ARA) kernel for the GPU. We describe how the blocked version of the algorithm is implemented and show the resulting performance on various sets of matrices together with comparison to SVD-based factorizations. Section 3 describes our algorithm for the ARA of a hierarchical matrix. The algorithm is formulated as a sequence of local low rank updates to a hierarchical matrix that gets constructed incrementally. We describe its three steps and their GPU implementation: the generation of the local low rank updates from a sampling strategy, the application of these updates to a hierarchical matrix structure, and the recompression of the hierarchical matrix to reduce the memory footprint of the approximation as it is being constructed. In section 4, we show the performance results of numerical experiments for the construction of a covariance matrix in a spatial statistics application, a hierarchical matrix-matrix multiplication, and a Schur complement matrix. We conclude with some remarks and future directions in section 5. The code developed for this work is available from the authors, together with scripts that allow for the performance results we describe to be reproduced.

1.1. Prior work.

Two general strategies have been proposed for the construction of hierarchically low rank matrices from matrix-vector operations: a top-down strategy and a bottom-up one. The top-down approach [20, 23] starts from the coarsest level of the matrix and "peels off" blocks from any given level by sampling with a random matrix that has a nonzero pattern specifically arranged to eliminate all but a particular block column. Once subspaces for these blocks are computed, a low rank approximation for them is generated and they are subtracted (peeled off) from the matrix. At the next level, the matrix is sampled with patterned random matrices and the contribution from the already factored coarser blocks is subtracted to provide the subspaces for block columns at that level. In [20] the pattern of sampling is obtained from the geometry of the grid that generated the hierarchical matrix, and the sampling pattern appropriate for a uniform cartesian grid is presented. More general grids, or hierarchical matrices with general partitioning, will require more careful patterned sampling and may decrease the amount of sampling that may be performed concurrently. In [23], an HSS representation ($\mathcal{H}^2$ with a weak admissibility pattern) is used. This allows all the sampling for a given level to be done concurrently in parallel, but the resulting matrix may have large ranks because of the fixed weak partitioning.

The algorithm we propose below is in this same vein but separates the sampling strategy from the target matrix being reconstructed. As we show below, we can sample using any convenient strategy and add low rank updates locally to the generally partitioned target matrix, while still keeping the nestedness of the basis. The separation of the structure sampling blocks from the structure of the matrix being constructed gives us flexibility to adapt the sampling strategy to the structure of the matrix and to the cost of matrix-vector sampling.

The bottom-up approach [22] starts from the finest level of the hierarchy and constructs a basis for that level, and then moves up the hierarchy. In order to construct a level basis from the sampled matrix, arbitrary entries of the input matrix must be available to allow low rank interpolative decompositions of blocks to be generated efficiently. The need for having explicit values for entries may limit the applicability of this factorization scheme. Xia [31] used a similar strategy for compressing blocks in a nested dissection solver, and the STRUMPACK package [27, 14] also uses this randomized sampling strategy for compression of fronts into HSS form. Shared and distributed memory hierarchical compression of SPD matrices into a weak admissibility format are described in [33, 34].

1.2. Notation and definitions.

To generate hierarchical matrix partitioning structures like the ones depicted in Figure 1, a hierarchical clustering tree of the index set of the rows and columns of the matrix is first constructed. A dual traversal of the cluster trees then produces pairs of index sets (t, s) representing blocks of the matrix. These pairs are tested against an admissibility condition that determines whether the block can be well approximated by a low rank matrix. Blocks that satisfy the condition become low rank leaves in the matrix tree and those that don't and are sufficiently large are subdivided further by resuming the dual tree traversal and become inner nodes of the matrix tree. Blocks that are determined to be small enough are kept in their original form as dense leaves of the matrix tree. The structures shown in Figure 1 are effectively the leaves of the matrix tree constructed by this dual traversal. Figure 2 shows the various levels of the matrix tree of the right general partitioning, where blocks corresponding to inner nodes are represented in blue, low rank leaves in green, and dense leaves in red. The blank blocks that appear in the third level onward are blocks that were not refined since they or their parent blocks satisfied the admissibility condition and became leaves in the tree. We will refer to a matrix block defined by the index set pair (t, s) of a matrix M at level l of the matrix tree as M^l_{ts}.

The $\mathcal{H}$-variant of hierarchical matrices stores the low rank form of an $m \times n$ matrix block of rank k directly in the leaves of the matrix tree in the form $M = AB^T$ or $M = USV^T$, where A and U are $m \times k$ matrices, B and V are $n \times k$ matrices, and S is a small $k \times k$ matrix. Assuming that k is much smaller than m and n, this

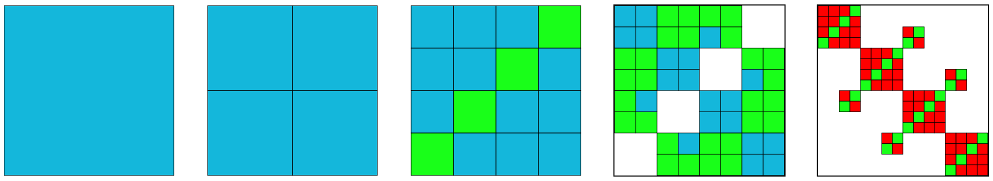


FIG. 2. *The levels of the matrix tree for a general partitioning of a matrix. The inner nodes are shown in blue, the low rank leaves in green, and the dense leaves in red.*

representation allows significant savings in memory and computation when compared to the original dense block. The $\mathcal{H}^2$-variant takes this representation further in two ways. First, all low rank leaves at the same level l of the matrix tree belonging to the same block row defined by the index set t share the same basis vectors U_t^l and those belonging to the same block column s share the same basis vectors V_s^l. Assuming a rank k^l at level l of the tree, this allows us to separate the row and column basis from the matrix tree into U and V basis trees and keep only small $k^l \times k^l$ coupling matrices S_{ts}^l in the leaves of the matrix tree. Each low rank leaf in that level can then be represented as $M_{ts}^l = U_t^l S_{ts}^l {V_s^l}^T$. The second difference lies in the way nodes are stored in the basis trees of a $\mathcal{H}^2$-matrix, where each inner node is not stored explicitly but expressed in terms of its children using small $k^l \times k^{l-1}$ transfer matrices to form a nested basis. Assuming a binary basis tree, a node $U_{t^+}^{l-1}$ can be expressed in terms of its child nodes $U_{t_1}^l$ and $U_{t_2}^l$ using their corresponding transfer matrices $E_{t_1}^l$ and $E_{t_2}^l$,

$$U_{t^+}^{l-1} = \begin{bmatrix} U_{t_1}^l & \\ & U_{t_2}^l \end{bmatrix} \begin{bmatrix} E_{t_1}^l \\ E_{t_2}^l \end{bmatrix}. \tag{1.1}$$

This ends at the leaves of the basis tree where each node is stored explicitly as a set of vectors.

Finally, we note that batched operations on GPUs are critical for the performance of algorithms that manipulate these trees. Requesting the execution of a kernel on a GPU causes a short stall before execution can begin that we call the kernel launch overhead. When the amount of work executed is small, this overhead can dominate the total runtime of the kernel. When operating on the small coupling and transfer matrices that are stored in the basis and matrix trees, we must first group as many operations that can be executed in parallel into a single batched operation that can then be carried out with a single kernel call. We call the grouping process that gathers all the necessary data for a batched kernel's execution from the basis and matrix trees *marshaling*, and we ensure efficient marshaling by flattening the aforementioned trees [6]. We denote data that has been marshaled and ready for batched execution by the $_{|||}$ subscript.

2. Adaptive randomized approximation.

An $m \times n$ matrix M is said to have low numerical rank when it can be well approximated by a low rank matrix of the form $M \approx AB^T$, where A and B are $m \times k$ and $n \times k$ matrices, respectively, and k is called the numerical rank. The numerical rank is typically much smaller than m and n, allowing computations with and storage of M to be more efficient as long as $k \leq \frac{mn}{m+n}$. This low rank approximation can be obtained in many ways, such as pivoted QR factorization, SVD, adaptive cross approximation [2], and many others. Randomized methods have also been developed [17] to compute the low rank approximation using only matrix-vector multiplication, making it an excellent choice

Algorithm 2.1. Fixed rank randomized approximation.

```
procedure FRA($M, k$)
  Input
    $M$   Matrix or matrix-vector black box
    $k$   Approximation rank
  Output
    $Q$   Left approximation factor
    $B$   Right approximation factor
  $[m, n] = size(M)$
  $\Omega = \text{rand}(n, k)$
  $Y = M\Omega$
  $[Q, \sim] = \text{qr}(Y)$
  $B = M^T Q$
```

for black box routines where matrix entries may not be available. While these methods have been used primarily for very large problems, we develop batched GPU routines for these algorithms that show substantial speedups when compared with the SVD for computing low rank approximations of large batches of small matrix blocks entirely on the GPU. In this section we will discuss some details of the algorithms and their implementation on the GPU.

2.1. Fixed rank. When the rank k of the matrix is known beforehand, the fixed rank randomized approximation method, described in Algorithm 2.1, can be used to compute the low rank approximation of an $m \times n$ matrix M. We first compute an approximate orthogonal basis Q for the column space of M as the left factor of the low rank approximation. This involves producing a random $m \times k$ sample matrix Y from the range of M by multiplying the matrix by a random $n \times k$ matrix Ω. The sample matrix can then be orthogonalized by performing a standard QR decomposition and discarding the triangular factor. It is worth noting that for matrices whose singular values decay rapidly, this method is sufficient to produce a good approximate basis Q; however, when the singular values decay slowly, further refinement of the basis by subspace iteration is required [17]. We do not discuss this enhancement to the algorithm here since the matrices we target fall in the former category. Once the approximate orthogonal basis Q has been determined, we compute the right factor B of the low rank approximation of $M \approx QB^T$ by simply forming the product $B = M^TQ$. If additional rank reduction is required (to satisfy some fixed error threshold), an approximate singular value decomposition can be easily obtained from this low rank form by computing the QR decomposition of B and then the SVD of the small $k \times k$ triangular factor. This method has been implemented as a batched GPU routine and used to accelerate the compression of hierarchical matrices [7] in place of the full singular value decomposition.

2.2. Adaptive rank. The rank of a matrix that satisfies a fixed precision is not typically known beforehand, making it difficult to use the fixed rank approximation in applications that have specific error tolerance requirements. Based on the adaptive randomized range finder algorithm [17], a fixed precision variant of the randomized approximation that adaptively determines the needed number of samples is described in Algorithm 2.2. It takes advantage of the fact that the basis can be constructed iteratively and the approximation error can be easily evaluated at each step. Instead

Algorithm 2.2. Adaptive randomized approximation.

```
procedure ARA($M$, $r$, $\epsilon$)
  Input
    $M$   Matrix or matrix-vector black box
    $r$   Required number of samples satisfying approximation error
    $\epsilon$   Approximation error
  Output
    $Q$   Left approximation factor
    $B$   Right approximation factor
  $[m, n] = \text{size}(M)$
  $Y = M \times rand(n, r)$                              ▷ Sample the matrix
  $Q = [\,]$                                             ▷ Initialize basis
  $j = 0, R_{max} = 0, rel_e = 1$
  while $rel_e > \epsilon / \left(10\sqrt{\frac{2}{\pi}}\right)$ and $j < n$ do
    $j = j + 1$
    $Y(:, j) = Y(:, j) - QQ^T Y(:, j)$                    ▷ Reprojection
    $Q(:, j) = Y(:, j)/\|Y(:, j)\|$                       ▷ Normalize
    $Y(:, j + r) = M \times rand(n, 1)$                   ▷ Take a new sample
    $Y(:, j + r) = Y(:, j + r) - QQ^T Y(:, j + r)$
    for $i = j + 1 : j + r - 1$ do
      $Y(:, i) = Y(:, i) - Q(:, j)Q(:, j)^T Y(:, i)$      ▷ Projection
    $abs_e = \max\{\|Y(:, j)\|, \ldots, \|Y(:, j + r)\|\}$
    $R_{max} = \max\{R_{max}, abs_e\}$
    $rel_e = abs_e/R_{max}$                               ▷ Update approximate relative error
  $B = M^T Q$
```

of producing all the samples at once, the orthogonal basis is generated by repeatedly sampling the matrix and orthogonalizing using a process similar to Gram–Schmidt (GS) with one reorthogonalization step [12, 29], until r consecutive projected samples satisfy the error threshold. The absolute error is approximated by the maximum norm of the last r sampled vectors, while the relative error is approximated by keeping track of the maximum norm of all projected sampled vectors. The relative error approximation is not a standard one but is easily obtained as a byproduct of the orthogonalization and performs well in practice. More complex methods, such as approximating the largest singular value, can be used to get a tighter relative error; however, in practice the additional computational cost is not worth any potential reduction in the computed rank.

2.3. GPU implementation. While the ARA can determine the appropriate rank for a specified error tolerance, most of the computations in Algorithm 2.2, such as the sampling at line 17, the projections at line 20, and the norm computations at line 21, are memory bound operations, while the computations in the fixed rank Algorithm 2.1 are mostly compute bound. When implemented on the GPU, a platform that greatly favors high arithmetic intensity, the fixed rank algorithm has a clear advantage in performance and blocked versions of randomized algorithms have been shown to have superior performance and better accuracy [24]. In this section, we will discuss the details of a blocked version of the ARA and its batched GPU implementation and compare its performance with the batched SVD routines developed in previous work [7].

2.3.1. Blocked version. Processing blocks of samples allows the use of BLAS Level 3 routines instead of the memory bound BLAS Levels 2 and 1 operations, greatly increasing the arithmetic intensity of the adaptive algorithm. The blocked algorithm is very similar to Algorithm 2.2; however, instead of sampling one vector at a time, the blocked algorithm produces multiple samples simultaneously and orthogonalizes using a blocked version of Gram–Schmidt (BGS) [29]. While this does come with the cost of potentially performing more work by oversampling, the performance gains more than compensate.

The original BGS algorithm still requires that blocks of columns Y be orthogonalized. This can be achieved using either the unblocked GS algorithm where high performance can be difficult to achieve due to the repeated norm computations or Householder QR factorization [7]. In the absence of efficient batched QR routines for the tall matrices produced by large problems, we turn to other methods that would allow us to make use of other high-performance routines. Taking advantage of the fact that we already perform one reorthogonalization step as well as the fact that the algorithm should stop sampling when the norms of the vectors reach the error threshold, we can use Cholesky QR [28] to perform the orthogonalization instead, allowing the use of much higher performance kernels. This involves computing a Gram matrix $G = Y^T Y$, computing the Cholesky decomposition of $G = R^T R$, and finally performing a triangular solve on the input columns $Y = YR^{-1}$ to produce the orthogonal columns. Cholesky QR is unfortunately numerically unstable due to the squaring of the condition number when forming the Gram matrix, causing the method to fail in single precision when a low relative error is needed. To overcome this, we compute the Gram matrix and the Cholesky factorization in mixed precision, reading in the data in single precision and computing in double precision. This gives us the benefits of the additional accuracy of double precision while keeping the original data in single precision and has been shown to greatly improve the numerical stability of the method [32]. When computing in double precision, we assume that the requested error threshold will not be extremely small, so we do not emulate quad precision [19, 21] and instead compute in the same precision as the input. In practice this has allowed us to compute with relative thresholds as low as 10^{-10}, significantly lower than the needs of our applications. If greater accuracy is required, quad precision may be needed to stabilize the double precision Cholesky QR.

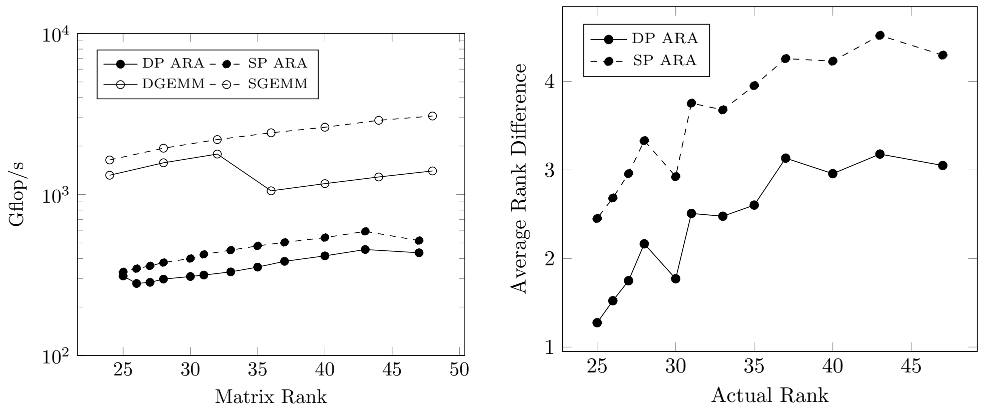
2.3.2. Implementation using batched routines. The nonuniform batched ARA GPU implementation of Algorithm 2.3 builds on several high-performance nonuniform batched kernels, specifically nonuniform matrix-matrix multiplication from the MAGMA library [25], random number generation from the CURAND library [26], as well as many kernels that we develop for this work which we discuss briefly here. To form the Gram matrices, we implement a batched mixed precision `syrk` GPU kernel that reads in the matrix data in the original precision and performs the computations in double precision. This kernel loads the matrix from global memory one block at a time into shared memory and stores and computes the output matrix in registers. It is specifically optimized for the case of transpose `syrk` of a tall and skinny matrix. We also implement a batched mixed precision `potrf` GPU kernel to compute the Cholesky factorization of the Gram matrices, outputting the triangular factor in the original precision while computing in double precision. Assigning multiple warps, where a warp is a group of 32 threads that execute instructions in parallel, to process multiple columns, the input matrix is loaded from global memory and factorized entirely in shared memory. This routine also stores and updates the diagonal of the

Algorithm 2.3. Batched adaptive randomized approximation.

```
procedure ARA($M$, $r$, $mr$, $bs$, $\epsilon$)    ▷ Acronym overloaded with unblocked routine
  Input
    $M$   Matrix batch or batch matrix-vector black box
    $r$   Required number of samples satisfying approximation error
    $mr$  Maximum allowed rank
    $bs$  Block size
    $\epsilon$   Approximation error
  Output
    $Q$   Left approximation factors
    $B$   Right approximation factors
  $[m, n, k] = \text{size}(M)$    ▷ Get batch and matrix size
  $Q = [\ ]$    ▷ Initialize basis
  $j = 0$    ▷ Current samples
  $br = \text{zeros}(k, 1)$    ▷ Rank of each matrix
  $svec = \text{zeros}(k, 1)$    ▷ "Small" vectors
  $c = 0$    ▷ Convergence
  $s = bs$    ▷ Block samples
  while $j < mr$ and $c \neq 1$ do
    $\Omega = \text{batchRand}(n, s, k)$
    $Y = \text{batchSample}(M, \Omega, k)$    ▷ Black box sample
    $dR = \text{ones}\ (bs, k)$    ▷ Diagonal of R
    for $i = 1 : 2$ do    ▷ BGS with one reorthog step
      $Z = \text{batchGemm}(Q^T, Y, k)$
      $Y = Y - \text{batchGemm}(Q, Z, k)$
      $G = \text{batchMpSyrk}(Y, k)$    ▷ Mixed precision Syrk
      $[R, dR] = \text{batchMpPotrf}\ (G, dR, k)$    ▷ Mixed precision Cholesky
      $Y = \text{batchTrsm}\ (Y, R, k)$
    $Q = [Q, Y]$
    $j = j + bs$
    $[svec, br] = \text{batchSetSvec}\ (dR, svec, br, r, bs, k, \epsilon)$    ▷ Check operation convergence
    $[c, s] = \text{batchSetSamples}(svec, r, bs, k)$    ▷ Check global convergence
  $B = M^T Q$
```

triangular factor as it goes through the reorthogonalization process since those values are used to determine the convergence of the operation (the equivalent of line 21 in Algorithm 2.2). These kernels can be easily extended to emulate quad precision input/output if additional accuracy is required. The kernel `batchSetSvec` determines whether an operation has converged by keeping track of the number of vectors with sufficiently small norms, while the kernel `batchSetSamples` determines how many samples to take for each operation in the batch as well as whether all operations have converged. Finally, sampling is done using a black box routine that is provided by the user and could be anything from a simple matrix-matrix multiplication to a specialized routine that only produces the output of multiple matrix-vector products.

2.3.3. Performance results. We show the effectiveness and performance of ARA when compared with the full SVD for generating an approximation of a batch of 1000 randomly generated numerically low rank matrices. The matrices are generated

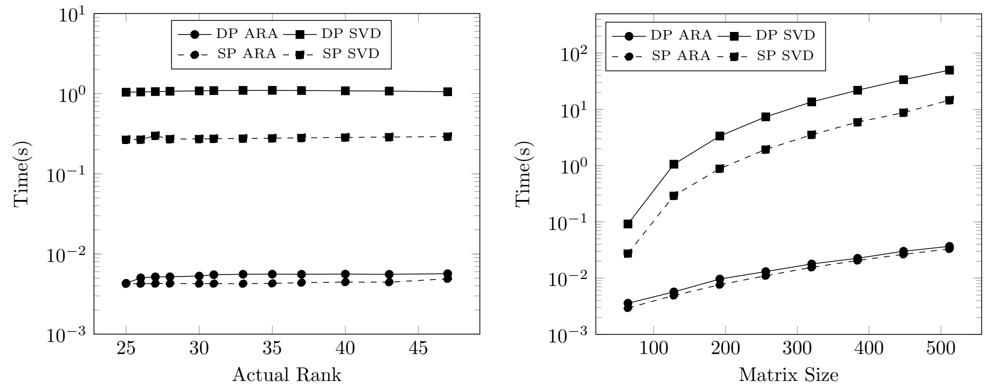


(a) Batched ARA performance in single and double precision for a batch of 1000 matrices of varying rank and fixed size (128×128). Performance of batched `gemm` included for reference.

(b) Average difference in numerical rank and detected rank in single and double precision for a batch of 1000 matrices.

FIG. 3. *Batched ARA performance for a precision of* 10^{-6}.

using the LAPACK routine `latms` and the singular values are generated using the expression $s_i = te^{-\alpha i}$ to simulate the expected rapid decay, where t is a random scaling factor and α controls the speed of the decay. Increasing α leads to faster singular value decay, producing different approximation ranks k for our chosen error threshold $\epsilon = 10^{-6}$. For the following results, we use a block size bs of 32 which was empirically determined to produce the best performance (not surprising since the warp size is 32) and r, the number of required consecutive small vectors for convergence, was set to 10. Figure 3(a) shows the performance of the batched ARA routine in GFLOP/s for a batch of 1000 dense 128×128 matrices for different values of α. ARA achieves a peak performance of 450 GFLOP/s in double and 600 GFLOP/s in single precision as opposed to the theoretical peak performance of 4.7 TFLOP/s in double and 9.3 TFLOP/s in single precision of the P100 GPU. In comparison, the batched-GEMM peformance on the P100 is 1.5 TFLOP/s in double and 3 TFLOP/s in single precision for a batch of 1000 matrix-matrix products of 128×128 matrices with $128 \times k$ matrices, which represent the practical peaks for the performance of the batched operations presented here. The flops for the batch are computed as the sum of operations performed for each matrix as determined by the unblocked Algorithm 2.2. The dip in performance from rank 43 to rank 47 is due to the need for an additional block of samples to satisfy the convergence criteria, producing a relatively large amount of unneeded samples. Figure 3(b) shows the average difference in numerical rank as determined by the truncation threshold ϵ and the decay of the singular values and the rank k detected by ARA for the aforementioned batch of matrices. The difference increases as the actual truncation rank increases due to slower singular value decay and can be reduced either by subspace iteration or by performing the SVD of the very small $k \times k$ matrices as discussed in section 2.1. It is worth noting that while the average difference in the rank for each batch may be relatively small, the batch may contain some matrices with a relatively large rank difference which may limit the method's usefulness in some situations. Figure 4(a) compares the time taken to compute the full SVD with the time for the ARA of the



(a) Time for the SVD and ARA of a batch of 1000 matrices in single and double precision with varying rank and fixed size (128×128).

(b) Time for the SVD and ARA of a batch of 1000 matrices in single and double precision with varying size and fixed rank (47).

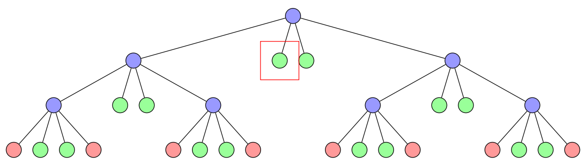
FIG. 4. *Comparison of batched ARA and SVD kernels.*

aforementioned batch of matrices. Since we compute the full SVD and then truncate, the time for each batch is almost the same and is up to 240x slower than the ARA. This is an expected result, since the blocked Jacobi SVD used here [7] is an $O(n^3)$ algorithm focusing on kernels that aren't particularly efficient while the ARA is $O(kn^2)$ for dense matrices of rank k and size $n \times n$ focusing on highly efficient matrix-matrix kernels. Figure 4(b) compares the times for a batch of 1000 matrices of various sizes and fixed rank of 47, highlighting the difference in complexity of the two algorithms.

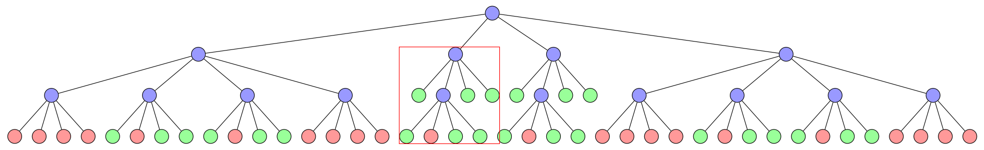
3. Hierarchical adaptive randomized approximation.

In this section, we discuss the details of a generalization of ARA that is used to construct a hierarchical representation of an $N \times N$ symmetric matrix using matrix-vector products in a process we call hierarchical adaptive randomized approximation (HARA). The method leverages the ideas and routines developed for the ARA to produce low rank approximations of blocks of a hierarchical matrix in a top-down manner. It is worth noting that while bottom-up algorithms [22] are asymptotically faster than top-down methods, differing by a factor of $O(\log N)$, they require the evaluation of the matrix entries in $O(1)$ time, a requirement that may not always be satisfied in applications.

The randomized approximation can be conceptually described as a two-phase process. In a first phase, a hierarchical matrix representation composed of low rank blocks is generated which is then, in a second phase, compressed into an $\mathcal{H}^2$ matrix with hierarchical bases. These two phases are not executed sequentially, however, but rather their applications are interleaved at every level. The low rank blocks generated in the first phase are accumulated level-by-level into the $\mathcal{H}^2$ matrix being constructed. There is never any need to store the complete hierarchical low rank matrix of the first phase, since it gets processed completely one level at a time. In order to perform the second phase accumulation by levels, the low rank blocks are turned into a set of local low rank updates that are then performed simultaneously on the whole matrix allowing high-performance batched kernels to be used in the accumulation and compression operations. If the ranks of the blocks generated in the first phase are too large to be processed in a single operation, the low rank updates can be split into smaller-sized updates and applied in sequence.



(a) Example of a sampling tree where a low rank representation of the highlighted node is first generated and then applied to the approximation tree below.



(b) Example of an approximation tree where a low rank update is applied to an inner node, resulting in updates to all the highlighted leaves of the subtree.

FIG. 5. *Examples of the sampling and approximation matrix trees and the application of a generated low rank update of the sampling tree to the approximation tree.*

We call the matrix tree structure that is used to determine which blocks are approximated by low rank constructions in the first phase the *sampling tree*. The matrix tree structure of the hierarchical matrix that is being constructed by the HARA is called the *approximation tree*. Examples of these two trees for the structures show in Figures 6 and 7 can be seen in Figure 5 and it is worth noting that while these two trees need not share the same structure, the approximation tree must be a more refined version of the sampling tree.

Unlike previous approaches to peeling [20, 23], our proposed algorithm decouples the sampling tree from the approximation tree. In [20], both the sampling and the target matrix use a standard admissibility structure, potentially requiring a large number of matrix-vector sampling operations, while in [23], both the sampling and the target use a weak admissibility structure, potentially resulting in a final matrix that has a large memory footprint because of large ranks in the off-diagonal blocks. The flexibility resulting from separating the sampling and approximation trees allows us to adapt the sampling structure to the structure of the final matrix and to the cost of matrix-vector sampling. It is often more effective to sample in blocks whose sizes are larger than the block sizes that are best for the memory footprint of the final hierarchical matrix. It is also often convenient to use a simple sampling strategy while refining the approximation tree to fit the specific patterns needed by an application. This ability to choose separately the sampling and approximation trees is possible because the construction algorithm is formulated as a sequence of local low rank updates of the target matrix.

The blocks that form the leaves of the sampling tree are sampled one level at a time by computing the product of the entire matrix with randomly generated structured vectors based on the sampling tree. The samples then undergo a process similar to the ARA to produce a set of low rank updates that are added into the constructed hierarchical matrix, affecting all leaves of the approximation tree that lie within the sampled blocks. Finally, to reduce the memory footprint of the constructed hierarchical matrix after the low rank updates, the matrix is recompressed. This leads us to Algorithm 3.1 for the HARA using only matrix-vector products on the input

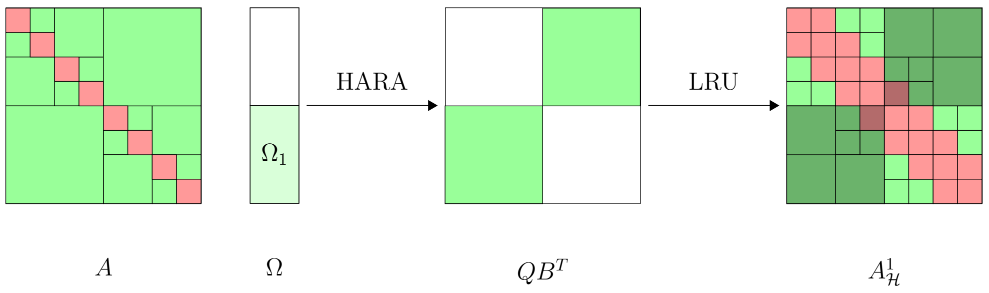


FIG. 6. *The major phases of one level of the HARA. The left figure shows the matrix A that is being approximated with the leaves of the sampling tree overlaid on it, as well as the structured random vectors generated by the sampling tree. These are used to generate the low rank updates of the center figure in a process similar to the ARA. Finally, the updates are applied to the hierarchical matrix $A_{\mathcal{H}}$ of the right figure, where the affected leaves of the matrix tree are highlighted in the shaded areas.*

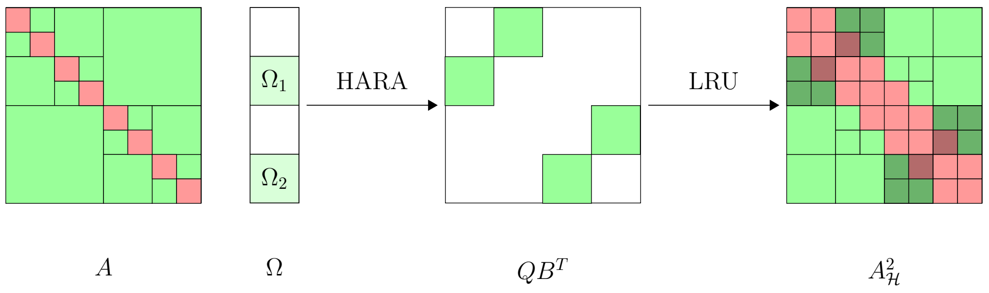


FIG. 7. *Sampling the second level of the matrix and updating the approximation tree nodes.*

matrix. Figures 6 and 7 show the overall process for two levels of the HARA process. The computational complexity of the sampling phase is $O(k \log N)$ times the cost per single sample, where k is an average rank of the sampled blocks at a given level. As we describe below, many of the sampling calls may be performed together on multiple samples and can result in substantial practical performance improvements: the number of multisample calls can be reduced to $O(\log N)$, i.e., a constant number of calls per level, each of which may be only a small factor more expensive that a single-sample call because of the higher arithmetic intensity involved. The accumulation and compression phase of the low rank updates of the construction operation has complexity $O(k^2 N \log N)$ and may or may not dominate depending on problem context and required accuracy which directly impacts k.

In the following sections we will discuss the algorithms used to generate the low rank updates and their application to the constructed hierarchical matrix as well as their implementation on the GPU.

3.1. Generating low rank updates. To approximate the blocks of the matrix that form the leaves of the sampling tree, we multiply the matrix by structured random vectors that allow us to sample those blocks without interference from other blocks. For example, if we need to sample the large upper right block of the simple sampling tree structure in Figure 6, we can generate a block of structured random vectors Ω where the top half that would be multiplied by the upper left block of the matrix is set to zero. The product of the matrix A with Ω is then computed as

Algorithm 3.1. Hierarchical adaptive randomized approximation.

```
procedure HARA($A$, $\mathcal{P}$, $A_{\mathcal{H}}$, $r$, $mr$, $bs$, $\epsilon$)
  Input
    $A$    Matrix or matrix-vector black box
    $\mathcal{P}$    Data or routines representing the sampling strategy
    $r$    Required number of samples satisfying approximation error
    $mr$   Maximum allowed rank
    $bs$   Block size
    $\epsilon$    Approximation error
  Output
    $A_{\mathcal{H}}$   Hierarchical matrix constructed from $A$
  $A_{\mathcal{H}} =$ zeroHmatrix($A$)
  for $l = 1 : levels(A_{\mathcal{H}})$ do
    $[U_{|||}, V_{|||}] =$ HARA-Level($A, \mathcal{P}, A_{\mathcal{H}}, l, r, mr, bs, \epsilon$)
    applyLRU($A_{\mathcal{H}}, \mathcal{P}, l, U_{|||}, V_{|||}$)      ▷ Apply low rank updates for level $l$
    $A_{\mathcal{H}} =$ hcompress($A_{\mathcal{H}}, \epsilon$)      ▷ Compress to the requested accuracy $\epsilon$
```

$$A\Omega = \begin{bmatrix} A_1^1 & R_1 \\ R_1^T & A_2^1 \end{bmatrix} \begin{bmatrix} 0 \\ \Omega_1 \end{bmatrix} = \begin{bmatrix} R_1\Omega_1 \\ A_2^1\Omega_1 \end{bmatrix}, \tag{3.1}$$

where R_1 is the leaf we want to sample, and A_1^1 and A_2^1 are diagonal nodes. Ignoring the lower block $A_2^1\Omega_1$, we can see that the matrix-vector product with the structured matrix Ω has given us the product $R_1\Omega_1$ of the block we need to sample with a set of random vectors, the key ingredient for ARA to produce the low rank approximation of the block. Although this process can be repeated for every leaf in the sampling tree, it will obviously not be efficient to do so, since sampling each block will involve a matrix-vector multiplication of the entire matrix A. Instead, we would like to be able to sample as many blocks as possible for each matrix-vector product that we perform.

3.1.1. Sampling strategy.

It is obvious that we can simultaneously sample all blocks within a block column s at level l by only filling in the entries of the random input vector corresponding to the index set of s; however, limiting ourselves to one block column per sample would lead to $O(n)$ growth in the number of needed matrix-vector products. When attempting to sample two blocks belonging to different block columns $R^l_{t_1s_1}$ and $R^l_{t_2s_2}$ at level l of the sampling tree using a single matrix-vector product, we must ensure that other blocks from the matrix do not interfere with the sampling, i.e., that there is no contribution to the output vector from matrix blocks in the intervals defined by (t_1, s_2) and (t_2, s_1). To simultaneously sample multiple blocks of a matrix, any interference must be dealt with using a combination of two sampling techniques.

The first technique sets to zero the entries of the random input vector that will interact with the unwanted blocks, as we have done in the previous section. The second technique sets the interfering blocks themselves to zero by subtracting or "peeling" them out from the input matrix. Since we don't have the exact representation of the blocks, we use the generated hierarchical matrix approximation instead. While this does introduce some errors into the approximation which could potentially accumulate as we go down the tree, we have observed in our experiments that the approximation error of the resulting hierarchical matrix satisfies the chosen error threshold. This could be a result of using a uniform rank for each level of the matrix, set to be the largest rank of all blocks at that level (due to the lack of necessary nonuniform batched GPU routines). The study and analysis of these errors and their effect on the choices for error thresholds at each level will be the focus of future work.

We peel the blocks of the matrix defined by the sampling tree in a top-down manner, generating approximations of all nodes one level at a time. This allows us to limit the search for interfering blocks to nodes that belong to the same level, since the nodes at a higher level are assumed to have already been peeled off. This leads us to the following general strategy: for each level l in the sampling tree, determine groups of blocks that can be sampled simultaneously assuming that the matrix blocks in the upper levels have been peeled off and generate structured random input vectors based on the index set of each group of blocks. The assumption that the higher levels have been peeled off is made valid by sampling the matrix $A^l = A - A^l_{\mathcal{H}}$, where $A^l_{\mathcal{H}}$ contains all the previously approximated levels. If we assume the simple weak admissibility structure of the symmetric matrix in Figure 6, we can simultaneously sample all the off-diagonal blocks at each level of the sampling tree with the same matrix-vector product. This can be seen in Figure 7, where all the off-diagonal blocks are sampled simultaneously. The dense diagonal blocks are extracted directly by multiplying A^l by a block column matrix where each block is the identity matrix.

More general sampling tree structures, i.e., peeling strategies, would need to efficiently determine which block column nodes can be grouped together based on whether all their respective nodes do not interfere with each other. A key observation here is that at every level the coupling matrix may be viewed as block sparse, where blocks corresponding to coarser levels have been eliminated. The problem of reducing the number of separate sampling calls can then be formulated as a graph coloring problem in the spirit of the CPR algorithm [11] for partitioning matrix columns in the computation of sparse Jacobians. The generation of a small number of groups of structurally orthogonal columns may be done using a greedy strategy as was done in [11], or by using more sophisticated heuristics in the solution of the graph coloring problem. Coleman, Garbow, and More [8], for example, write the problem as a distance-1 graph coloring problem. In this formulation, a column intersection graph of a matrix is used where a column in the matrix corresponds to a vertex, and an edge exists between two vertices if their corresponding columns have nonzeros in a common row. In our matrix peeling problem, structurally nonorthogonal column blocks would correspond to having blocks not yet peeled in a common block row, and this is done at every level of the matrix hierarchy. For symmetric matrices, path coloring [9] may be used to generate more effective groups of structurally orthogonal columns. A detailed survey and discussion of this general problem appears in [13]. This will be the focus of future work.

3.1.2. GPU implementation. Algorithm 3.2 details the generation of the low rank updates for a level of the matrix. It is structurally very similar to the ARA algorithm, needing only a few changes made to the sampling process and low rank approximation management. We first define our sampling strategy $\mathcal{P}$ as a set of data or routines that determine which blocks can be sampled simultaneously based on the structure of the sampling tree. In the simple case of a sampling tree having a weak admissibility structure, the sampling strategy $\mathcal{P}$ would simply return all the off-diagonal blocks in the level l in a single batch. This is due to the fact that for the simple structure generated by weak admissibility, all off-diagonal blocks within a level can be sampled at the same time without interference. As mentioned in section 3.1.1, more complex sampling strategies for standard admissibility structures would

Algorithm 3.2. Hierarchical adaptive randomized approximation.

```
procedure HARA-Level($A$, $\mathcal{P}$, $A_{\mathcal{H}}$, $l$, $r$, $mr$, $bs$, $\epsilon$)
  Input
    $A$    Matrix or matrix-vector black box
    $\mathcal{P}$    Data or routines representing the sampling strategy
    $A_{\mathcal{H}}$   Hierarchical matrix constructed from $A$ up to level $l-1$
    $l$    Current sampling level
    $r$    Required number of samples satisfying approximation error
    $mr$   Maximum allowed rank
    $bs$   Block size
    $\epsilon$    Approximation error
  Output
    $U_{|||}$   Left factors of low rank updates generated by HARA
    $V_{|||}$   Right factors of low rank updates generated by HARA
  $U = V =$ zeros $(A.n, bs)$
  $[k, U_{|||}, V_{|||}] =$ marshalLRU$(\mathcal{P}, l, U, V)$          ▷ Marshal low rank updates
  $j = 0$                                                   ▷ Current samples
  $br =$ zeros$(k, 1)$                                        ▷ Rank of each block
  $svec =$ zeros$(k, 1)$                                      ▷ "Small" vectors
  $c = 0$                                                   ▷ Convergence
  $s = bs$                                                  ▷ Block samples
  while $j < mr$ and $c \neq 1$ do
    $\Omega_{|||} =$ marshalRandomInput$(\mathcal{P}, l)$
    batchRand$(\Omega_{|||}, k)$    ▷ Generate structured random input using marshaled data
    $Y =$ sampleLevel$(A, A_{\mathcal{H}}, \Omega)$                       ▷ Black box sample
    $dR =$ ones $(bs, k)$                                     ▷ Diagonal of R
    for $i = 1 : 2$ do                                 ▷ BGS with one reorthog step
      $Z =$ batchGemm$({U_{|||}}^T, Y, k)$
      $Y = Y -$ batchGemm$(U_{|||}, Z, k)$
      $G =$ batchMpSyrk$(Y, k)$                            ▷ Mixed precision Syrk
      $[R, dR] =$ batchMpPotrf $(G, dR, k)$                ▷ Mixed precision Cholesky
      $Y =$ batchTrsm $(Y, R, k)$
    $U = [U, Y]$
    $j = j + bs$
    $[svec, br] =$ batchSetSvec $(dR, svec, br, r, bs, k, \epsilon)$ ▷ Check operation convergence
    $[c, s] =$ batchSetSamples$(svec, r, bs, k)$            ▷ Check global convergence
  $U_{|||} =$ marshalClearInput$(U, \mathcal{P}, l)$
  batchZeroBlock$(U_{|||})$                              ▷ Clear out marshaled U blocks
  $V =$ sampleLevel$(A, A_{\mathcal{H}}, U)$                           ▷ Black box sample
```

potentially produce multiple batches of blocks that can be sampled simultaneously, but the basic idea remains the same, where only the contents and size of the batched blocks would change. Pointer and dimension data for the low rank updates U and V are marshaled for the current level l based on $\mathcal{P}$. A simplified version of this marshaling routine is shown in Algorithm 3.3, where pointer data is generated from the indexes that define the blocks generated by $\mathcal{P}$. In a similar manner, the random input vector is generated by first marshaling the necessary pointer and dimension

Algorithm 3.3. Marshaling low rank update data.

```
procedure MARSHALLRU($\mathcal{P}$, $l$, $U$, $V$)
  Input
    $\mathcal{P}$    Data or routines representing the sampling strategy
    $l$    Current sampling level
    $U$    Output block of vectors for the left factors
    $V$    Output block of vectors for the right factors
  Output
    $U_{|||}$    Pointer data for the left factors of low rank updates
    $V_{|||}$    Pointer data for the right factors of low rank updates
  $k = \text{blockCount}\ (\mathcal{P}, l)$        ▷ Blocks in $l$ that can be sampled simultaneously
  for $i = 1 : k$ do
    $[i_1, i_2, j_1, j_2] = \text{block}\ (\mathcal{P}, l, i)$        ▷ Get the indexes defining the block
    $U_{|||i} = \text{ptr}\ (U) + i_1$
    $V_{|||i} = \text{ptr}\ (V) + j_1$
```

data from $\mathcal{P}$ and then passing the data to a batched block random number generator (implemented using device level CURAND routines). Sampling is handled by a black box routine that performs the matrix-vector multiplication of the matrix $A^l = A - A^l_{\mathcal{H}}$ with the generated random vectors. This can obviously be split into the original black box routine and a regular hierarchical matrix-vector product. Finally, the right factor V is produced by first marshaling the necessary data from the left factor U using $\mathcal{P}$, clearing the data using a batched routine, and multiplying A^l by U.

3.2. Applying low rank updates. After generating the low rank approximations of the leaves of the sampling tree, they must be added to the constructed hierarchical matrix as low rank updates while maintaining the nested basis property of the basis tree. As mentioned in the previous section, the sampling and approximation trees do not need to have the same structure, making it possible to add low rank updates to inner nodes of the approximation tree. These updates can be performed in linear complexity [5] even for matrices with general partitioned structure, as long as the number of blocks per row is bounded at every level in the hierarchy, a condition that is naturally satisfied by matrices that admit hierarchical representations. In this section, we discuss the details of the algorithm used to apply low rank updates to the nodes of the approximation tree as well as its GPU implementation.

3.2.1. Updating the basis nodes. Given a low rank update $R_i = Q_i B_i^T$ to a node A^l_{ts} of the approximation tree, we must update the corresponding nodes U^l_t and V^l_s in the basis tree while maintaining the nested basis property discussed in section 1. Assuming the matrix is symmetric and that for every low rank update $R_i = Q_i B_i^T$ there must be a corresponding low rank update $R_i^T = B_i Q_i^T$ to maintain the symmetry of the matrix, we will focus on the changes to the U basis tree. The nonsymmetric case follows the same algorithm with updates to the V basis tree.

If U_t is a leaf node, the update to the basis is trivial; we simply append the columns of Q_i to the columns of the explicitly stored basis node:

$$\bar{U}^l_t = \begin{bmatrix} U^l_t & Q_i \end{bmatrix}, \tag{3.2}$$

where $\bar{U}^l_t$ is the new basis node. If U_t is an inner node which is only stored implicitly by the transfer matrices of its children, the update Q_i is split into block rows and applied

Algorithm 3.4. Applying low rank updates to a hierarchical matrix.

```
procedure APPLYLRU($A_\mathcal{H}, \mathcal{P}, l_u, U_{|||}, V_{|||}$)
  Input
    $A_\mathcal{H}$  Hierarchical matrix under construction
    $\mathcal{P}$  Data or routines representing the sampling strategy
    $l_u$  Updated level
    $U_{|||}$  Left factors of low rank updates generated by HARA
    $V_{|||}$  Right factors of low rank updates generated by HARA
  Output
    $A_\mathcal{H}$  Hierarchical matrix is updated
  $[E_{|||}, F_{|||}]$ = flagBasisNodes ($A_\mathcal{H}, \mathcal{P}, l_u$)        ▷ Marshal updated basis nodes
  $d = levels(A_\mathcal{H})$
  for $l = l_u : d$ do
    updateTransferNodes ($A_\mathcal{H}, E_{|||}, F_{|||}, l$)
  updateBasisLeaves ($A_\mathcal{H}, E_{|||}, F_{|||}, d, U_{|||}, V_{|||}$)
  $[S_{|||}, D_{|||}]$ = flagCouplingNodes ($A_\mathcal{H}, \mathcal{P}, l_u$)     ▷ Marshal updated matrix tree nodes
  for $l = l_u : d$ do
    updateCouplingNodes ($A_\mathcal{H}, S_{|||}, l$)
  updateDenseNodes ($A_\mathcal{H}, D_{|||}, U_{|||}, V_{|||}$)
```

recursively to its children. If node t has two children c_1 and c_2, we can represent the new node as

$$\bar{U}_t^l = \begin{bmatrix} U_t^l & Q_i \end{bmatrix} = \begin{bmatrix} U_{c_1}^{l+1} E_{c_1}^{l+1} & Q_{i_1} \\ U_{c_2}^{l+1} E_{c_2}^{l+1} & Q_{i_2} \end{bmatrix} = \begin{bmatrix} U_{c_1}^{l+1} & Q_{i_1} & 0 & 0 \\ 0 & 0 & U_{c_2}^{l+1} & Q_{i_2} \end{bmatrix} \begin{bmatrix} E_{c_1}^{l+1} & 0 \\ 0 & I \\ E_{c_2}^{l+1} & 0 \\ 0 & I \end{bmatrix}, \tag{3.3}$$

where Q_{i_1} and Q_{i_2} are the block rows of Q_i corresponding to the index sets of the children nodes c_1 and c_2, respectively. The above equation shows us that passing on the block rows to the children and replacing the transfer matrices of the node by block diagonal matrices, where the upper block is the old transfer matrix and the lower block is the identity matrix, allows us to implicitly append the columns of the update to the columns of the basis node. Finally, to maintain the nested basis property, the rows of the transfer matrices of the parent node which was not affected by the update are padded with zeros. For example, the parent basis node t^+ for an updated node t_1 whose sibling is t_2 is represented as

$$U_{t^+}^{l-1} = \begin{bmatrix} U_{t_1}^l & 0 \\ 0 & U_{t_2}^l \end{bmatrix} \begin{bmatrix} E_{t_1}^{l+1} \\ E_{t_2}^{l+1} \end{bmatrix} = \begin{bmatrix} U_{t_1}^l & Q_i & 0 \\ 0 & 0 & U_{t_2}^l \end{bmatrix} \begin{bmatrix} E_{t_1}^{l+1} \\ 0 \\ E_{t_2}^{l+1} \end{bmatrix}. \tag{3.4}$$

3.2.2. Updating the coupling nodes. After the basis is properly updated, we must update the coupling matrices of the hierarchical matrix. If the updated node A_{ts}^l is a leaf in the approximation tree, the update simply involves replacing the coupling matrix of the node with a block diagonal matrix where the upper block is the original coupling matrix and the lower block is the identity:

$$A_{ts}^l + Q_i B_i^T = U_t^l S_{ts}^l V_s^l + Q_i B_i^T = \begin{bmatrix} U_s^l & Q_i \end{bmatrix} \begin{bmatrix} S_{ts}^l & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} V_t^l & B_i \end{bmatrix} = \bar{U}_t^l \bar{S}_{ts}^l \bar{V}_s^l, \tag{3.5}$$

where $\bar{S}_{ts}$ is the new coupling node. If the updated node is an inner node, then the aforementioned update is carried out on all of its children. If a child node happens to be a dense block, then the update is carried out explicitly using matrix-matrix multiplication.

3.2.3. GPU implementation. Algorithm 3.4 describes the major steps of the GPU implementation of applying the low rank update to a level of the hierarchical matrix. To efficiently implement the application of low rank updates on the GPU, we again split the operation into a marshaling phase where pointer and dimension data is generated and an execution phase using high-performance batched GPU routines. The low rank updates are generated only as matrix data in the form $R_i = Q_i B_i^T$ and the index i of the node in the sampling tree corresponding to the approximated block. Since the sampling and approximation trees can be different, the first step determines the node indexes of the approximation tree corresponding to each sampling tree node index in the update list. This can be done very efficiently using the `lower_bound` Thrust routine with the Morton order index of each node within the current level used as the search key to search for all node indexes simultaneously with a single kernel call. Using the approximation tree node index data of the updates, we flag the affected nodes in the basis tree and then use a single transformation Thrust kernel per level of the tree to flag the children of the nodes. Flagged inner nodes are updated by first copying the original transfer matrices using a batch block copy kernel followed by a batched kernel that sets the lower block diagonal to the identity matrix as discussed in section 3.2.1. The flagged leaf nodes are updated using two batched copy kernels for the original leaf data and the appropriate block rows of the low rank update. The coupling matrices are updated similarly, where updated nodes are flagged in one pass and the children are flagged one level at a time. Flagged dense blocks are updated using batch matrix-matrix multiplication kernels from CUBLAS.

3.3. Compression to reduce rank. To maintain the optimal complexity of the hierarchical operations and reduce the memory consumption of the hierarchical matrix after ranks increase due to the low rank updates, the matrix must be compressed. Compression involves computing a new compact basis for the blocks of the matrix followed by a projection of the blocks into the new basis. The details of the algorithm are described in [6]. We briefly summarize the major phases of the compression here in order to highlight an optimization to the algorithm that can be employed when low rank updates are applied top-down one level at a time as we do in our construction procedure.

The algorithm first orthogonalizes the bases. A basis is orthogonalized by performing a QR decomposition at the leaves of the basis and sweeping up the tree to orthogonalize the inner nodes using the nested basis property:

$$U_{t^+}^{l-1} = \begin{bmatrix} U_{t_1}^l & \\ & U_{t_2}^l \end{bmatrix} \begin{bmatrix} E_{t_1}^l \\ E_{t_2}^l \end{bmatrix} = \begin{bmatrix} \dot{U}_{t_1}^l & \\ & \dot{U}_{t_2}^l \end{bmatrix} \begin{bmatrix} T_{t_1}^l E_{t_1}^l \\ T_{t_2}^l E_{t_2}^l \end{bmatrix} = \begin{bmatrix} \dot{U}_{t_1}^l & \\ & \dot{U}_{t_2}^l \end{bmatrix} Z_{t^+}, \tag{3.6}$$

where $\dot{U}$ is the new orthogonal basis and the T matrices are the triangular factors of the QR decompositions of the basis nodes. A QR decomposition is then performed on each generated Z matrix to produce the T matrices of the current level. The appropriate block rows of the orthogonal factor replace the transfer matrices of the node, finalizing the orthogonalization of the node. The T matrices are then used in a projection phase to update the coupling matrices:

$$U_t^l S_{ts}^l V_s^{lT} = \dot{U}_t^l \left(T_t^l S_{ts}^l T_t^{lT} \right) \dot{V}_s^{lT} = \dot{U}_t^l \dot{S}_{ts}^l \dot{V}_s^{lT}. \tag{3.7}$$

Once the basis is orthogonalized, we generate an optimal basis from the data of the matrix in a downsweep pass (details are omitted here, but see, e.g., [6]) and compress the basis in an operation that is structurally very similar to the orthogonalization, using the SVD of the nodes instead of the QR decomposition and truncating the singular values that are smaller than our truncation threshold ϵ. These operations typically sweep up and down the entire tree; however, in the context of compression after low rank updates, one can save a significant amount of computation by taking advantage of the fact that we update in a top-down manner, where the levels above the updated level remain untouched. Consider an orthogonal basis tree that has been updated at level l. To reorthogonalize the basis, we proceed with the regular algorithm until we reach level l. We know that the nodes above that level remain untouched by the update and therefore are still orthogonal. For an orthogonal node $U_{t^+}^{l-1}$, the matrix Z_{t^+} from (3.6) must be orthogonal, since the block diagonal matrix $\left[\begin{smallmatrix} \dot{U}_{t_1}^l & \\ & \dot{U}_{t_2}^l \end{smallmatrix}\right]$ is also orthogonal. Orthogonalizing the matrix Z_{t^+} in this case would simply produce the identity matrix as its triangular factor T, resulting in no changes to the basis nodes at higher levels or their associated coupling matrices. Therefore, the process can stop at this level by directly replacing the transfer matrices $E_{t_1}^l$ and $E_{t_2}^l$ with the block rows $T_{t_1}^l E_{t_1}^l$ and $T_{t_2}^l E_{t_2}^l$, respectively. Similarly, the compression process can be sped up by starting the optimal basis downsweep at the updated level, since the matrix data of the upper levels did not change, and the upsweep can be cut short with similar reasoning to the orthogonalization.

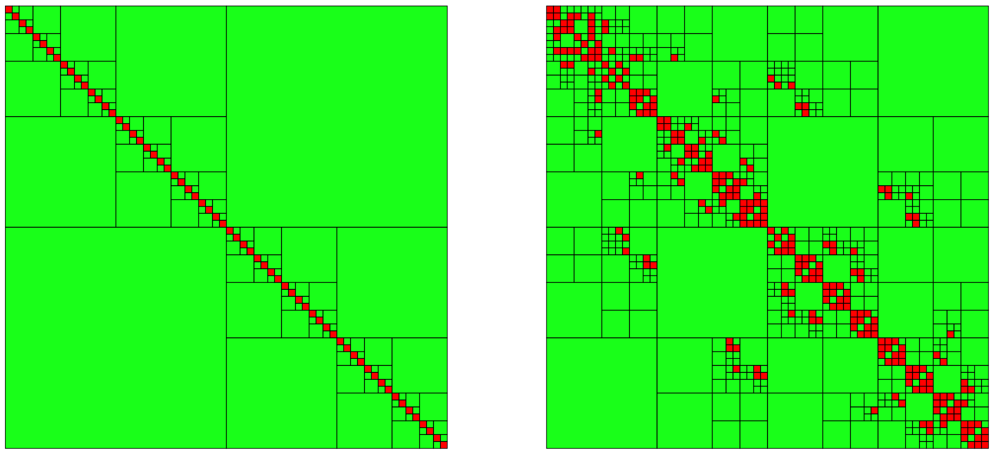
4. Numerical experiments.

To analyze the performance of the HARA algorithm, we present results from three tests. The first one is a self-reconstruction experiment aiming to reconstruct approximations of a given hierarchical matrix with different accuracy thresholds by accessing it only through matrix-vector products. The second test is a matrix-matrix multiplication operation, where we aim to construct the product by sampling it via two matrix-vector operations involving the matrix operands. The third test computes the Schur complement of a discretization of a Poisson operator. All experiments were run on a 16 GB Pascal P100 GPU and performance results were averaged over 10 runs.

Even though matrix-vector operations involving single vectors are fast, both in the $O(N)$ asymptotic sense and in practical running times on GPUs [6], they are bandwidth-limited computations and therefore cannot optimally exploit GPU capabilities. We have implemented a kernel that allows multiple matrix-vector products to be performed simultaneously to allow for increased arithmetic intensity and avoid bandwidth-limited bounds on performance. We use this kernel in the first two experiments.

4.1. Construction of a spatial statistics covariance matrix.

For the first experiment, we generate covariance matrices for a two-dimensional (2D) spatial Gaussian process with a spatially varying density of point distribution and an isotropic exponential kernel with correlation length 0.1. Hierarchical representations of the formally dense $N \times N$ covariance matrices are formed analytically by first clustering the points in a KD-tree using a mean split giving us the hierarchical index sets of the basis trees. The basis vectors and transfer nodes are generated using Chebyshev interpolation [4]. The approximation tree is constructed using a dual traversal of the basis tree [15], and the coupling matrices are generated by evaluating the kernel at


(a) The leaves of the weak admissibility sampling tree for a problem size of 2^{12}.

(b) The leaves of the matrix approximation tree for a problem size of 2^{12}.

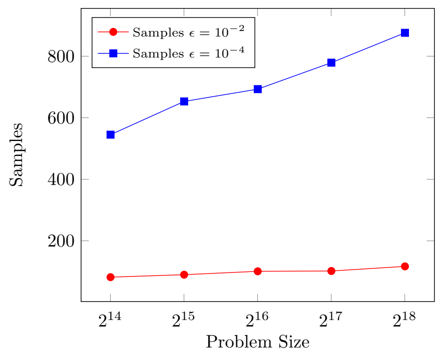
FIG. 8. *Sampling and approximation trees used to construct a hierarchical matrix.*

the interpolation points. The matrix is constructed to a relative accuracy of 10^{-7} in the Frobenius norm.

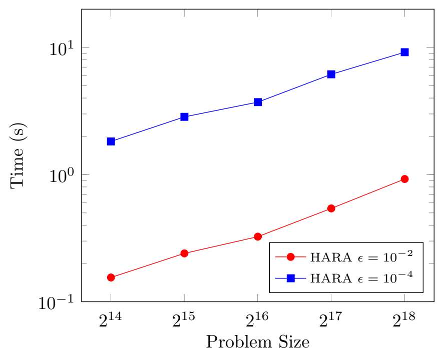
In this experiment, the black box sampling matrix-vector multiplication routine is simply the product of the hierarchical covariance matrix with the input vector. The constructed hierarchical matrix has a leaf size of 64 and a refined matrix tree shown in Figure 8(b). The sampling tree used was the simple weak admissibility sampling structure shown in Figure 8(a). We used a block size bs of 32 and two approximation thresholds for HARA at 10^{-2} and 10^{-4} to study the effect of the error threshold on the runtime as well as on the computed ranks and samples. To verify that the constructed hierarchical matrix $A_{\mathcal{H}}$ achieves the requested accuracy, we approximate the difference between $A_{\mathcal{H}}$ and the black box A by taking the maximum relative error of the product of the difference with a set of 10 normalized random vectors x_i:

$$\bar{\epsilon} = \max_{i=1\ldots10} \frac{\|(A_{\mathcal{H}} - A)\, x_i\|}{\|Ax_i\|}. \tag{4.1}$$

Figure 9(a) shows the sum of all samples taken for each level of the hierarchical matrix. The tighter error threshold clearly requires significantly more samples to approximate the matrix and exhibits faster growth in the number of samples as the problem size increases. This is reflected in the overall runtimes of the algorithm for each threshold, as seen in Figure 9(b), where the approximation with the higher accuracy take almost 7x the time to complete when compared to the lower accuracy approximation. For comparison, our single-threaded reference implementation of the compression on a 2.4-GHz Broadwell CPU for a problem size of 2^{14} is completed in about 12 seconds for an accuracy of 10^{-2} and 370 seconds for an accuracy of 10^{-4}. Figure 10 breaks down the computation into its four major phases. We note that most of the time is spent in the sampling phase for the lower accuracy approximation but in the compression phase for the higher accuracy. This difference is primarily due to the significantly larger ranks in the constructed hierarchical matrix as shown in Figure 11, where the runtimes of the relatively inefficient SVDs used in the compression dominate those of the extremely efficient matrix multiplications of the sampling.



(a) Number of samples required to construct an $\mathcal{H}^2$ matrix representation to a specified accuracy.



(b) GPU time for the construction.

FIG. 9. *Scalability and performance of HARA on a representative covariance matrix from spatial statistics.*

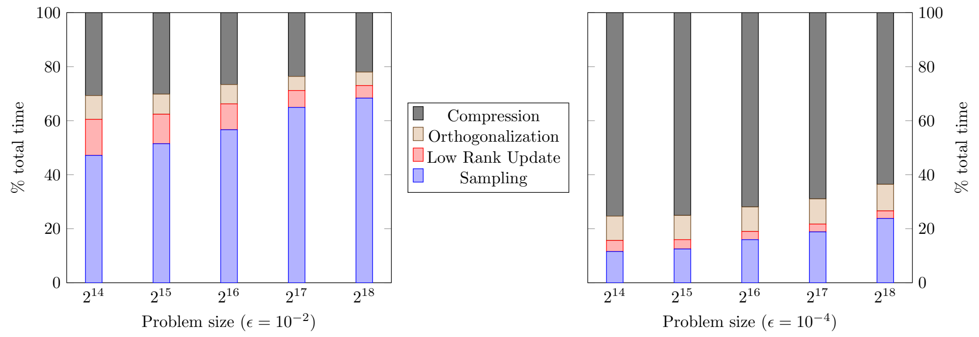


FIG. 10. *Profile showing relative costs of the different phases of the reconstruction operation.*

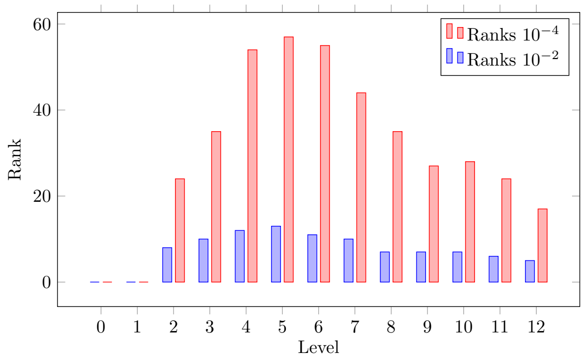


FIG. 11. *Comparison of the ranks per level for a problem size of* 2^{18} *for error thresholds of* 10^{-2} *and* 10^{-4}.

Figure 12 shows the achieved performance of the overall construction operation. It is possible to further optimize the GPU kernels inside the compression operations by using a randomized SVD following batched ARA approximations, but this requires

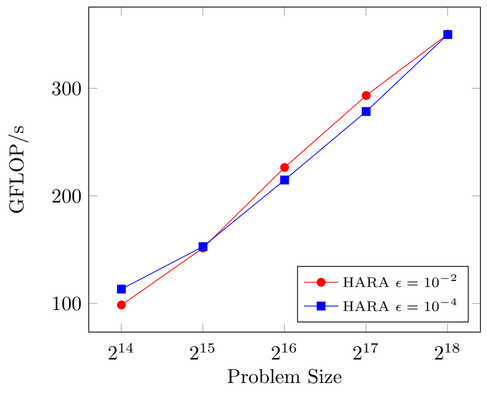


FIG. 12. *Overall performance for the construction of the hierarchical matrix on a P100 GPU.*

some additional nonuniform batched routines in our software infrastructure, and we leave this as future work.

4.2. Hierarchical matrix-matrix multiplication. In this second experiment, we show that the HARA construction routine can perform a matrix-matrix multiplication operation by constructing the square of a hierarchical matrix. We use a covariance matrix similar to the one in the previous section with a randomly perturbed point set on a regular $[0, 1]^2$ grid.

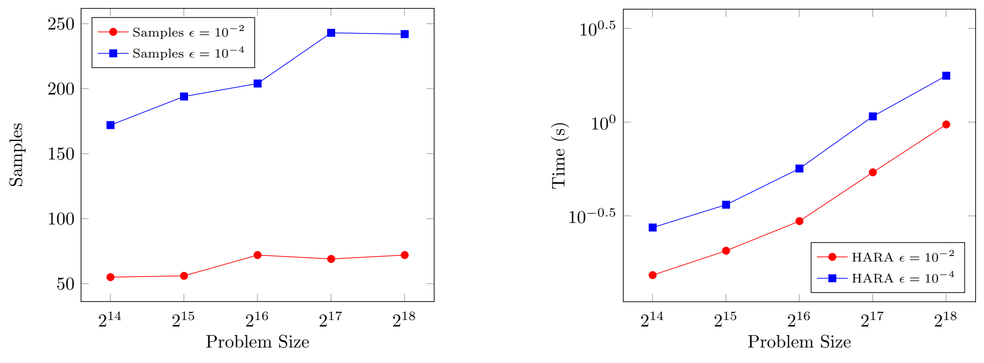
This experiment illustrates the ability of the method to compute a general matrix expression, given only the ability to multiply the expression by a vector. The matrix squaring operation also allows us to examine the effect of the faster decay of singular values that accompanies the squaring on the performance of the algorithm. In a randomized matrix-matrix multiplication operation, the number of samples needed depends on the ranks of the blocks in the resulting product and not on the ranks of the operands.

The black box sampling routine in this case simply performs two hierarchical matrix-vector multiplications using an intermediate vector z to store the first product:

$$y = A^2 x = A\,(Ax) = Az.$$

We use the HARA algorithm and the same hierarchical matrix parameters for A as in the first experiment. As expected, the number of required samples for both 10^{-2} and 10^{-4} error thresholds decreases in comparison to the first experiment. There is also a slower growth in the required number of samples for the lower error threshold case as shown in Figure 13(a). Figure 13(b) shows that the overall gap in runtime between the lower and higher accuracy thresholds is also significantly lower for this problem, requiring only as much as twice the time for the increased accuracy. Figure 14 verifies our expectation that the ranks at each level of the constructed hierarchical matrix are also significantly reduced, leading to a heavier emphasis on the sampling phases for both thresholds. The accumulation and compression phases take up a smaller amount of the total computation as a result, as shown in Figure 15, and lead to greater overall performance for the higher accuracy as seen in Figure 16 due to the increased arithmetic intensity in the more efficient sampling phase.

4.3. Schur complement computation. In this experiment, we show how HARA may be used to construct a Schur complement matrix, which is formally dense,



(a) Number of samples required. (b) GPU time for the construction.

FIG. 13. *Scalability and performance of HARA for constructing an $\mathcal{H}^2$ representation of A^2.*

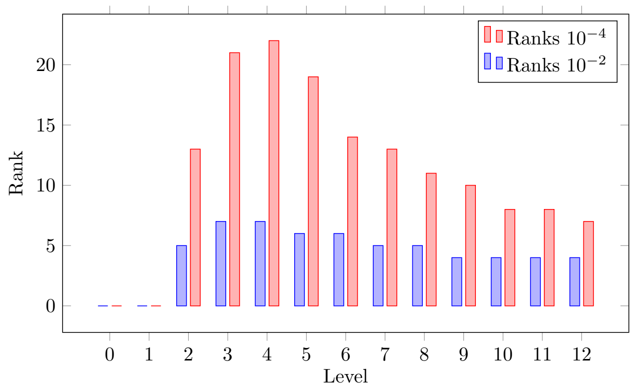


FIG. 14. *Comparison of the ranks per level when constructing the square of the matrix for a problem size of 2^{18} for error thresholds of 10^{-2} and 10^{-4}.*

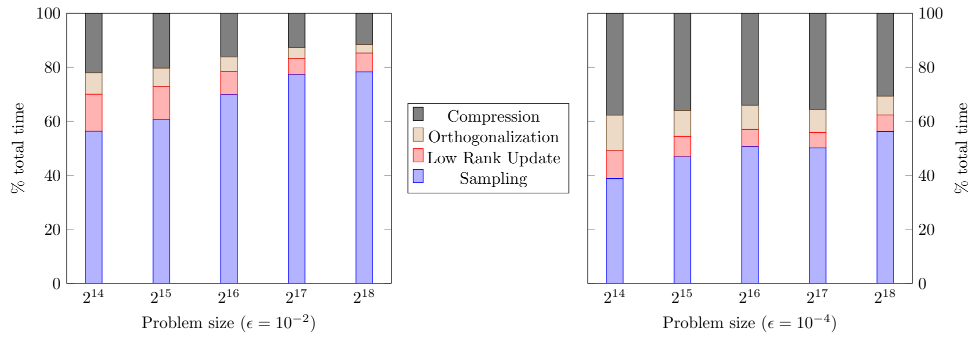


FIG. 15. *Profile showing relative costs of the different phases of the multiplication operation.*

directly as a hierarchical matrix. We consider 5-pt (in two dimensions) and 7-pt (in three dimensions) discretizations of a constant coefficient Poisson operator on a uniform cartesian grid and partition the resulting sparse matrix as

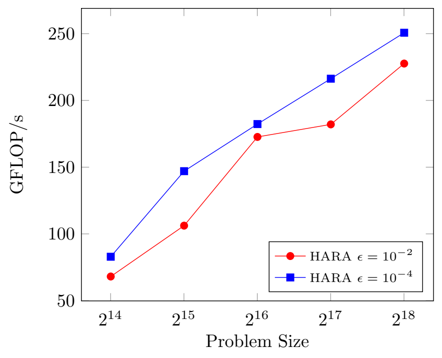


FIG. 16. *Overall performance for the construction of the square of an $\mathcal{H}^2$ matrix on a P100.*

TABLE 1
Results for computing the Schur complement to an accuracy of $\epsilon = 10^{-4}$.

Mesh size	Size of S	# of Samples	Sampling time (s)	Compression time (s)	Memory (MB) / compression
1024×1024	1024^2	34	14.1	0.029	0.46 / 17.4
2048×2048	2048^2	42	39.8	0.039	0.90 / 35.5
$32 \times 32 \times 32$	1024^2	152	3.52	0.103	2.59 / 3.1
$64 \times 64 \times 64$	4096^2	397	140.7	0.817	20.0 / 6.4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where “2” denotes the nodes that correspond to one boundary edge of the grid in two dimensions or boundary plane in three dimensions, and “1” refers to the remaining nodes. We seek to compute the Schur complement of the 11 block. Computing Schur complements is a core computation in many factorization strategies, e.g., [18], and we plan to build on it in future work to develop complete solvers for GPUs.

Given a factorization of $A_{11} = L_{11}L_{11}^t$ on the GPU, the CUPARSE library can compute the product of the Schur complement $S = A_{22} - A_{12}(L_{11}L_{11}^t)\backslash A_{21}$ with vectors, via sparse matrix-vector products and sparse triangular forward and backward solves. This sequence of operations provides the black box sampling operation needed for the construction of S. For a 2D $n \times n$ grid, the size of S is $n \times n$, while the triangular solves are roughly of size $n^2 \times n^2$. For a 3D $n \times n \times n$ grid, the size of S is $n^2 \times n^2$, while the triangular solves are roughly of size $n^3 \times n^3$. We used a weak admissibility structure for the approximation matrix in two dimensions and a slightly more refined structure with a standard admissibility condition for the 3D problem. All computations were done in double precision.

Table 1 lists the sampling and compression results of constructing the Schur complement for 2D and 3D problems of various sizes, to an accuracy of $\epsilon = 10^{-4}$. The sampling time dominates the total construction time, and more specifically the triangular solve portions of the sampling are the operations that dominate the total runtime, as triangular solvers are difficult to parallelize on the GPU. This bottleneck exists despite the fact that the triangular solvers of the CUSPARSE library benefit greatly from the increased arithmetic intensity and parallelism enabled by the blocking of the samples. For example, the sampling time was reduced to 14 seconds from 177 seconds for the 2D 1024×1024 problem, a speedup of over 12x over an unblocked, one vector at a time, sampling.

As expected, the 2D problems have significantly smaller local ranks and therefore require fewer samples and show greater compression than the 3D problems, where compression is measured as the memory footprint of the full dense representation of S (not constructed) over its hierarchical counterpart. In both cases the compression time without sampling, which includes generation and application of the low rank updates, was a negligible portion of the total runtime.

5. Conclusions.

In this paper we presented GPU algorithms for two important problems in low rank matrix approximations. The first is a high-performance batched ARA kernel for the generation of low rank factorization of a set of numerically low rank matrices. The kernel requires access to the matrices only via matrix-vector products. It processes multiple samples simultaneously and orthogonalizes them using a blocked version of Gram–Schmidt. The algorithm is built on top of a batched mixed precision Cholesky QR that allows us to reach high performance while enhancing the numerical stability of Cholesky QR. The ARA kernel allows nonuniform matrix sizes in the batch to allow more flexibility for applications.

The second algorithm targets multilevel hierarchical matrices, where every level in the matrix hierarchy is composed of a large set of small low rank blocks. In addition these low rank blocks are expressed in terms of nested bases, where row and column bases for the blocks at a given level can be computed in terms of the bases of the next finer level. This $\mathcal{H}^2$ representation allows the storage of compressible matrices in optimal memory complexity, and our algorithm can generate this optimal representation of a given matrix to a specified target accuracy, requiring access to the matrix via matrix-vector products only. The algorithm operates in two stages interleaved level by level. In the first stage randomized low rank approximations of the blocks of a given level are generated, and in a second stage these blocks are accumulated into the taget matrix as local low rank updates and compressed into optimal nested bases. The local low rank updates are performed simultaneously to allow us to achieve high performance. The algorithm can apply the low rank updates for a given level in stages to avoid excessive memory usage during the construction procedure. Overall, the algorithm requires $O(k \log N)$ matrix-vector products to be computed and $O(k^2 N \log N)$ computations to generate the $N \times N$ matrix, where k is an average rank for the matrix blocks. In addition, the matrix-vector products may be done in blocks of bs vectors to boost parallelism and arithmetic intensity. In some contexts, the cost of computing of the product of a matrix by multiple vectors is only marginally more expensive than a single matrix-vector multiplication, which greatly enhances the performance of the first phase of the algorithm. One of the immediate applications of this hierarchical matrix construction is the matrix-matrix multiplication operation, or more generally the construction of any algebraic matrix expression whose product with a vector can be computed. Numerical experiments confirm the effectiveness and performance of the algorithms.

The hierarchical matrix construction algorithm described and the current implementation accompanying this paper target symmetric matrices, but extensions to the nonsymmetric case are straightforward.

REFERENCES

[1] M. Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, www.tensorflow.org/about/bib (2011).
[2] M. Bebendorf and S. Rjasanow, *Adaptive low-rank approximation of collocation matrices*, Computing, 70 (2003), pp. 1–24.

[3] S. Börm, *Efficient Numerical Methods for Non-Local Operators: $\mathcal{H}^2$-Matrix Compression, Algorithms and Analysis*, EMS Tracts Math. 14, European Mathematical Society, Zürich, 2010.

[4] S. Börm and J. Garcke, *Approximating Gaussian processes with $\mathcal{H}^2$-matrices*, in European Conference on Machine Learning, Springer, New York, 2007, pp. 42–53.

[5] S. Börm and K. Reimer, *Efficient arithmetic operations for rank-structured matrices based on hierarchical low-rank updates*, Comput. Vis. Sci., 16 (2013), pp. 247–258.

[6] W. Boukaram, G. Turkiyyah, and D. Keyes, *Hierarchical matrix operations on GPUs: Matrix-vector multiplication and compression*, ACM Trans. Math. Software, 45 (2019), pp. 3:1–3:28.

[7] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, *Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression*, Parallel Comput., 74 (2018), pp. 19–33.

[8] T. F. Coleman, B. S. Garbow, and J. J. More, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[9] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Software for estimating sparse Hessian matrices*, ACM Trans. Math. Software, 11 (1985), pp. 363–377.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, *Torch7: A MATLAB-like environment for machine learning*, presented at Neural Information Processing Systems, 2011.

[11] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, IMA J. Appl. Math., 13 (1974), pp. 117–119.

[12] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization*, Math. Comput., 30 (1976), pp. 772–795.

[13] A. Gebremedhin, F. Manne, and A. Pothen, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Rev., 47 (2005), pp. 629–705.

[14] P. Ghysels, X. Li, F. Rouet, S. Williams, and A. Napov, *An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. S358–S384.

[15] L. Grasedyck and W. Hackbusch, *Construction and arithmetics of $\mathcal{H}$-matrices*, Computing, 70 (2003), pp. 295–334.

[16] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, Springer, New York, 2015.

[17] N. Halko, P. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.

[18] S. Hao and P.-G. Martinsson, *A direct solver for elliptic pdes in three dimensions based on hierarchical merging of Poincaré–Steklov operators*, J. Comput. Appl. Math., 308 (2016), pp. 419–434.

[19] Y. Hida, X. S. Li, and D. H. Bailey, *Algorithms for quad-double precision floating point arithmetic*, in Proceedings of the 15th IEEE Symposium on Computer Arithmetic, IEEE, 2001, pp. 155–162.

[20] L. Lin, J. Lu, and L. Ying, *Fast construction of hierarchical matrix representation from matrix–vector multiplication*, J. Comput. Phys., 230 (2011), pp. 4071–4087.

[21] M. Lu, B. He, and Q. Luo, *Supporting extended precision on graphics processors*, in Proceedings of the Sixth International Workshop on Data Management on New Hardware, ACM, 2010, pp. 19–26.

[22] P. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 1251–1274.

[23] P. Martinsson, *Compressing rank-structured matrices via randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. A1959–A1986.

[24] P. Martinsson and S. Voronin, *A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices*, SIAM J. Sci. Comput., 38 (2016), pp. S485–S507.

[25] R. Nath, S. Tomov, and J. Dongarra, *Accelerating GPU kernels for dense linear algebra*, in Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, Berkeley, CA, Springer, New York, 2010.

[26] *cuRAND Library Programming Guide*, Version 8.0, NVIDIA, 2018, https://docs.nvidia.com/cuda/curand.

[27] F. Rouet, X. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, ACM Trans. Math. Software, 42 (2016), pp. 27:1–27:35.

[28] A. Stathopoulos and K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM J. Sci. Comput., 23 (2002), pp. 2165–2182.

[29] G. W. Stewart, *Block Gram–Schmidt orthogonalization*, SIAM J. Sci. Comput., 31 (2008), pp. 761–775.

[30] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Comput., 36 (2010), pp. 232–240.

[31] J. Xia, *Randomized sparse direct solvers*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 197–227.

[32] I. Yamazaki, S. Tomov, and J. Dongarra, *Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs*, SIAM J. Sci. Comput., 37 (2015), pp. C307–C330.

[33] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, *Geometry-oblivious FMM for compressing dense SPD matrices*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 53:1–53:14.

[34] C. D. Yu, S. Reiz, and G. Biros, *Distributed-memory hierarchical compression of dense SPD matrices*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, 2018, pp. 15:1–15:15.