Solving Linear Systems on a GPU with Hierarchically Off-Diagonal Low-Rank Approximations

Chao Chen
Oden Institute for Computational Engineering and Sciences
University of Texas at Austin
Austin, United States
chenchao.nk@gmail.com

Per-Gunnar Martinsson
Department of Mathematics
Oden Institute for Computational Engineering and Sciences
University of Texas at Austin
Austin, United States
pgm@oden.utexas.edu

***Abstract*—We are interested in solving linear systems arising from three applications: (1) kernel methods in machine learning, (2) discretization of boundary integral equations from mathematical physics, and (3) Schur complements formed in the factorization of many large sparse matrices. The coefficient matrices are often data-sparse in the sense that their off-diagonal blocks have low numerical ranks; specifically, we focus on "hierarchically off-diagonal low-rank (HODLR)" matrices. We introduce algorithms for factorizing HODLR matrices and for applying the factorizations on a GPU. The algorithms leverage the efficiency of batched dense linear algebra, and they scale nearly linearly with the matrix size when the numerical ranks are fixed. The accuracy of the HODLR-matrix approximation is a tunable parameter, so we can construct high-accuracy fast direct solvers or low-accuracy robust preconditioners. Numerical results show that we can solve problems with several millions of unknowns in a couple of seconds on a single GPU.**

***Index Terms*—Linear solver on GPU, boundary integral equation, kernel matrix, elliptic partial differential equations, hierarchical low-rank approximation, batched dense linear algebra, rank structured matrix, hierarchical matrix, LU factorization.**

I. Introduction

Consider the solution of a large dense linear system

$$A\,x = b, \quad A \in \mathbb{F}^{N\times N}, x \text{ and } b \in \mathbb{F}^{N}, \tag{1}$$

on a GPU, where $\mathbb{F}$ is the field of real or complex numbers. We focus on the situation where the coefficient matrix A can be approximated by *hierarchically off-diagonal low-rank* (HODLR) matrices [1], [2]. Given a matrix X partitioned into a 2×2 block form:

$$X = \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}, \tag{2}$$

we say X is a HODLR matrix if (1) the two off-diagonal blocks X_{12} and X_{21} are low rank, and (2) the two diagonal blocks X_{11} and X_{22} have the same off-diagonal low-rank structure or have sufficiently small sizes. (See a pictorial illustration in Fig. 2.)

Solving (1) on a GPU using the HODLR-matrix approximation is particularly appealing for two reasons. (1) The approximation reduces the required memory footprint and thus allows solving much larger problem sizes than storing the entire matrix A. For example, we were able to solve problems with several millions of unknowns on a single GPU that has only 32 GB of memory (see, e.g., Table IV). (2) Our algorithms are built upon the batched matrix-matrix multiplication and the batched LU factorization routines, which are highly efficient on GPUs. For example, the construction of our GPU solver achieved approximately 2 TFlop/s on an NVIDIA V100 GPU (see Fig. 9).

HODLR matrices arise from a range of applications across science, engineering, and data analytics, including:

a) Kernel matrices: Given a (real) kernel function $\mathcal{K}$, such as the Gaussian kernel and the Matern kernel, and a data set $\{y_i\}_{i=1}^{N}$, the associated kernel matrix $K \in \mathbb{R}^{N\times N}$ is defined as

$$K_{i,j} = \mathcal{K}(y_i, y_j), \qquad \forall i, j = 1, 2, \ldots, N.$$

Such matrices arise in machine learning [3], [4] and data assimilation [5], [6]. Ambikasaran et al. [7], [8] demonstrated that these matrices can be approximated efficiently by HODLR matrices. See also our numerical results in section IV-A.

b) Boundary integral equations: Some boundary value problems (BVPs) involving, e.g., the Laplace and the Helmholtz equations, can be reformulated as boundary integral equations (BIEs) of the form

$$a(x)u(x) + \int_{\Gamma} K(x, y)u(y)dy = f(x) \tag{3}$$

where $K(\cdot, \cdot)$ is derived from the free-space fundamental solution associated with the elliptic operator; where $a(\cdot)$ and $f(\cdot)$ are given functions; and where $u(\cdot)$ is the unknown. While it may be challenging to discretize the original partial differential equations (PDEs) directly, the BIEs offer several advantages (see, e.g., [9, Chapter 10]). Although the discretization of (3) leads to a dense linear system, the discretized integral operator can be approximated efficiently by a HODLR matrix in many environments. See sections IV-B and IV-C for two concrete examples.

c) Elliptic PDEs: The discretization of an elliptic PDE

$$-\nabla \cdot (a(x)\nabla u(x)) + b(x)u(x) = f(x),$$

where $a(\cdot)$, $b(\cdot)$, and $f(\cdot)$ are given functions, leads to a *sparse* linear system to be solved. While sparse direct solvers [10] based on Gaussian elimination are extremely robust, they typically require significant computing resources to handle dense Schur complements arising during the elimination. That significant acceleration can be attained by exploiting rank-structures in these Schur complements was established in, e.g., [2], [11], [12].

A. Previous work

Classical direct methods for solving (1) based on LU or QR factorizations admit highly efficient implementations on GPUs, but are limited by their unfavorable $\mathcal{O}(N^3)$ scaling in terms of operations. For GPU computing, the $\mathcal{O}(N^2)$ storage complexity can be even more restrictive.

This paper concerns a class of methods that exploit the numerically low-rank property of off-diagonal blocks in the matrix A. As many theoretical and empirical results have demonstrated, certain off-diagonal blocks in A can be compressed efficiently by their low-rank approximations. Based on this observation, the tile low-rank (TLR) approximation [13], [14] views matrix A as a block matrix and compresses off-diagonal blocks using low-rank approximations. With the TLR approximation, the LU factorization or the Cholesky factorization of matrix A can be accelerated significantly. This approach has been realized efficiently on shared and distributed-memory systems for covariance matrices [15] and BIEs [16]. It is also implemented on GPUs recently [17]. However, the asymptotic complexity for solving (1) is generally super-linear and the cost can be significant when the matrix size N is large.

To arrive at linear or nearly linear complexities, a hierarchical decomposition of matrix A is needed, and several hierarchically low-rank approximations have been proposed including $\mathcal{H}$ matrices [18], [19], $\mathcal{H}^2$ matrices [20], hierarchically semi-separable (HSS) matrices [21]–[23] and HODLR matrices. See related algorithms for solving (1) in, e.g., [5], [8], [9], [24]–[32] and the references therein. However, few of these methods have been implemented to achieve high performance on modern computing architectures. Two recent software packages for HODLR matrices include (1) HODLRLIB [33], which is an open-source code written in C++ and parallelized with OpenMP for multicore CPUs, and (2) hm-toolbox [34], which is a MATLAB code that implements many operations involving HODLR matrices focusing on prototyping algorithms and ensuring reproducibility rather than delivering high performance.

B. Contributions

We introduce a new data structure for HODLR matrices, where the low-rank bases of all off-diagonal blocks are concatenated into two big matrices (see an example in Fig. 3). This strategy aggregates small sub-blocks in the bases that need to be processed during the factorization into large sub-blocks in the big matrices. As a result, we can combine sequences of BLAS/LAPACK calls of low arithmetic intensities into a single kernel launch, which increases the Flop rate and mitigates the overhead of data transfer. This approach naturally leads to (1) a high-performance factorization algorithm that delivered up to 20 GFlop/s on a single CPU core in our experiments, and (2) parallel algorithms for factorizing a HODLR matrix and for applying the factorization to solve linear systems. Furthermore, we implemented the parallel algorithms leveraging existing high-performance batched matrix-matrix multiplication and batched LU factorization routines on a GPU. We present numerical benchmarks for solving large dense linear systems arising from kernel methods and two dimensional BIEs, and we compare the performance of our methods to existing methods/codes.

II. PRELIMINARILES

We adopt the following MATLAB notations: (1) $A(\mathcal{I},:)$ and $A(:,\mathcal{I})$ denote the rows and columns in matrix A corresponding to an index set $\mathcal{I}$, respectively; and (2) $[\,A\,|\,B\,]$ denotes the concatenation of two matrices A and B that have the same number of rows. Matrices and vectors are denoted using upper and lower case letters, respectively. In particular, I is used to denote an identity matrix of an appropriate size. We use greek letters to denote nodes in a tree data structure.

A. HODLR matrix

Here we give (non-recursive) definitions of a cluster tree and the associated HODLR format, which follow closely with [9], [34]. A cluster tree stands for a hierarchical partitioning of the row/column indices of a matrix, which dictates a tessellation of the matrix in a HODLR format.

Definition 1 (Cluster tree)**.** *Given an index set $\mathcal{I} := \{1, 2, \ldots, N\}$, a cluster tree $\mathcal{T}_L$ is a binary tree that satisfies the following three conditions:*

1) *There are $L+1$ levels, namely, $0, 1, \ldots, L$, and there are 2^ℓ nodes at level ℓ.*
2) *Every tree node stands for a (nonempty) consecutive subset of $\mathcal{I}$. In particular, the root node represents $\mathcal{I}$.*
3) *The union of two subsets owned by a pair of siblings, respectively, equals to the subset owned by their parent. (Nodes at the same level form a partitioning of $\mathcal{I}$.)*

Fig. 1 shows an example of $\mathcal{T}_2$. In practice, the indices in $\mathcal{I}$ are usually associated with points in $\mathbb{R}^k$, so $\mathcal{T}_L$ can be computed using some recursive bisection strategies. For example, $\mathcal{T}_L$ can be constructed as a k-d tree. Notice that a cluster tree $\mathcal{T}_L$ naturally leads to a tessellation of a matrix $A \in \mathbb{F}^{N\times N}$:

1) A leaf node α corresponds to a diagonal block $A(\mathcal{I}_\alpha, \mathcal{I}_\alpha)$, and there are 2^L of them.
2) A pair of siblings α and β corresponds to an off-diagonal block $A(\mathcal{I}_\alpha, \mathcal{I}_\beta)$, and there are $2^{L+1} - 2$ of them.

For example, Fig. 2 illustrates such a tessellation.

Definition 2 (HODLR matrix)**.** *Given a cluster tree $\mathcal{T}_L$, a matrix $A \in \mathbb{F}^{N\times N}$ is a HODLR matrix if every off-diagonal*

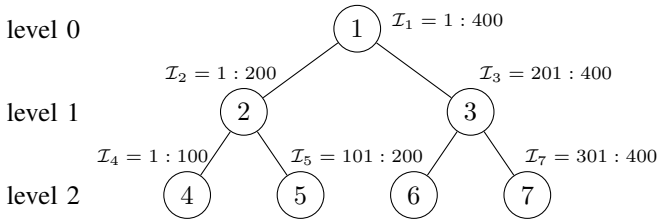


Fig. 1: An example of hierarchical decomposition of matrix row/column indices $\mathcal{I} = \{1, 2, \ldots, 400\}$. The root is the entire set $\mathcal{I}_1 = \mathcal{I}$, and there are four leaf nodes in this case. Some of the individual index vectors are given. Node 2 has two "children" 4 and 5, who are "siblings".

$$A = \begin{array}{|cc|cc|} \hline D_4 & A_{4,5} & \multicolumn{2}{c|}{\multirow{2}{*}{$A_{2,3}$}} \\ A_{5,4} & D_5 & & \\ \hline \multicolumn{2}{|c|}{\multirow{2}{*}{$A_{3,2}$}} & D_6 & A_{6,7} \\ \multicolumn{2}{|c|}{} & A_{7,6} & D_7 \\ \hline \end{array}$$

Fig. 2: Matrix tessellation corresponding to the cluster tree in Fig. 1. A diagonal block D_α is defined as $A(\mathcal{I}_\alpha, \mathcal{I}_\alpha)$, where α is a leaf node.

block $A(\mathcal{I}_\alpha, \mathcal{I}_\beta)$ that corresponds to a pair of siblings α and β in $\mathcal{T}_L$ has low rank. We say a HODLR matrix has rank r if the maximum rank of all off-diagonal blocks is r.

B. Construction of a HODLR matrix

The construction of a HODLR matrix is straightforward in the sense that we need to only compress certain off-diagonal blocks in the original matrix. We refer interested readers to Ambikasaran's PhD thesis [5] for a review of algebraic and analytic techniques to compute the low-rank approximations. See also [35] for some recent advances on randomized methods. In situations where a fast matrix-vector product routine exists for the matrix to be compressed, the so-called "peeling algorithms" [36]–[38] have been developed for the construction of a HODLR approximation.

Several parallel algorithms have been developed to construct more complicated formats than HODLR matrices, and some of them can be used to construct a HODLR-matrix approximation. Fernando et al [39] demonstrates the construction of an HSS matrix on multiple GPUs using an almost matrix-free method introduced in [40]. Boukaram et al [41] introduces a matrix-free method to construct an $\mathcal{H}^2$ matrix on a GPU. Chenhan et al [42], [43] develop algorithms on multicore and distributed-memory machines to approximate a symmetric positive definite (SPD) matrix with hierarchically low-rank structures.

III. Algorithms

Assume a rank-r HODLR matrix $A \in \mathbb{F}^{N \times N}$ and the underlying cluster tree $\mathcal{T}_L$ are given. Our focus here is factorizing the HODLR matrix and applying the factorization to solve a linear system

$$A\,x = b. \tag{4}$$

Let every off-diagonal block $A(\mathcal{I}_\alpha, \mathcal{I}_\beta)$ (that corresponds to a pair of siblings in $\mathcal{T}_L$) be that

$$A(\mathcal{I}_\alpha, \mathcal{I}_\beta) = U_\alpha V_\beta^*, \tag{5}$$

where U_α and V_β are two skinny matrices, and we call them the left and the right low-rank bases.

A. Recursive algorithm

Consider a partitioning of (4) into

$$\begin{pmatrix} A_\alpha & U_\alpha V_\beta^* \\ U_\beta V_\alpha^* & A_\beta \end{pmatrix} \begin{pmatrix} x_\alpha \\ x_\beta \end{pmatrix} = \begin{pmatrix} b_\alpha \\ b_\beta \end{pmatrix}, \tag{6}$$

where α and β are the two nodes at level 1 in $\mathcal{T}_L$. Suppose we can solve the following two subproblems with multiple right-hand sides:

$$\left\{ \begin{array}{l} A_\alpha\, z_\alpha = b_\alpha \\ A_\alpha\, Y_\alpha = U_\alpha \end{array} \right. \quad \text{and} \quad \left\{ \begin{array}{l} A_\beta\, z_\beta = b_\beta \\ A_\beta\, Y_\beta = U_\beta \end{array} \right. . \tag{7}$$

Then, the solution of (4) can be obtained via

$$\begin{pmatrix} x_\alpha \\ x_\beta \end{pmatrix} = \begin{pmatrix} z_\alpha \\ z_\beta \end{pmatrix} - \begin{pmatrix} Y_\alpha w_\alpha \\ Y_\beta w_\beta \end{pmatrix}, \tag{8}$$

where w_α and w_β are from solving

$$\begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix} \begin{pmatrix} w_\alpha \\ w_\beta \end{pmatrix} = \begin{pmatrix} V_\alpha^* z_\alpha \\ V_\beta^* z_\beta \end{pmatrix}. \tag{9}$$

Obviously, (7) can be solved recursively when α and β are not leaves in $\mathcal{T}_L$; otherwise, we solve (7) directly using, e.g., the LU factorization. This recursive algorithm initially appeared in [2], and we prove its correctness below.

Theorem 1 (Correctness of recursion)**.** *Assume the following three matrices are invertible:*

$$A_\alpha, \quad A_\beta, \quad \textit{and} \quad \begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix}.$$

Then, (8) *is the solution of* (6)*.*

Proof. Multiply $\begin{pmatrix} A_\alpha^{-1} & \\ & A_\beta^{-1} \end{pmatrix}$ on both sizes of (6), and it is sufficient to prove

$$\begin{pmatrix} I & Y_\alpha V_\beta^* \\ Y_\beta V_\alpha^* & I \end{pmatrix} \begin{pmatrix} x_\alpha \\ x_\beta \end{pmatrix} = \begin{pmatrix} z_\alpha \\ z_\beta \end{pmatrix}.$$

According to the Woodbury formula, we know that

$$\begin{aligned} \begin{pmatrix} I & Y_\alpha V_\beta^* \\ Y_\beta V_\alpha^* & I \end{pmatrix}^{-1} &= \left[I + \begin{pmatrix} & Y_\alpha \\ Y_\beta & \end{pmatrix} \begin{pmatrix} V_\alpha^* & \\ & V_\beta^* \end{pmatrix} \right]^{-1} \\ &= I - \begin{pmatrix} & Y_\alpha \\ Y_\beta & \end{pmatrix} \begin{pmatrix} I & V_\alpha^* Y_\alpha \\ V_\beta^* Y_\beta & I \end{pmatrix}^{-1} \begin{pmatrix} V_\alpha^* & \\ & V_\beta^* \end{pmatrix}. \end{aligned} \tag{10}$$

Therefore, it is straightforward to verify that (8) holds. □

Observe that some calculations in the recursion do not depend on the right-hand side b, and thus intermediate results can be precomputed. In particular, we divide the computation into two stages:

- Factorization: we compute Y_α, Y_β, and an LU factorization of
$$K_\gamma \triangleq \begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix}, \tag{11}$$
where γ is the parent of α and β.
- Solution: we compute z_α, z_β, w_α, w_β, x_α, and x_β.

Notice that some of the calculations can be done in-place (overwriting inputs with outputs). For example, we can overwrite two U matrices with corresponding Y matrices. As a result, the extra memory required by the algorithm mostly comes from storing the LU factorization of the coefficient matrix and the right-hand sides in (9), which is negligible when the rank in (5) is small.

B. Data structure and nonrecursive algorithm

For simplicity, let us assume the rank of all off-diagonal blocks is r. (The following presentation extends to general cases where the ranks vary.) To motivate the new data structure we use in our parallel algorithms, we illustrate the main idea through the following example.

Example 1. *Consider the HODLR matrix in Fig. 2, where the underlying cluster tree has three levels (level 0, 1, and 2). An appropriate partitioning of* (4) *leads to*

$$\begin{pmatrix} A_2 & U_2 V_3^* \\ U_3 V_2^* & A_3 \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \end{pmatrix},$$

where A_2 and A_3 are themselves (2-level) HODLR matrices; $b_2 = b(\mathcal{I}_2)$ and $b_3 = b(\mathcal{I}_3)$ are subvectors in the right-hand side b; and $x_2 = x(\mathcal{I}_2)$ and $x_3 = x(\mathcal{I}_3)$ are subvectors in the solution x.

Let us write down the recursion steps. At the first step, we have two subproblems:

$$\begin{cases} A_2\, z_2 = b_2 \\ A_2\, Y_2 = U_2 \end{cases} \quad \textit{and} \quad \begin{cases} A_3\, z_3 = b_3 \\ A_3\, Y_3 = U_3 \end{cases}.$$

Rewrite them more compactly as

$$A_2\,[\, z_2 \,|\, Y_2 \,] = [\, b_2 \,|\, U_2 \,] \quad \textit{and} \quad A_3\,[\, z_3 \,|\, Y_3 \,] = [\, b_3 \,|\, U_3 \,],$$

which again can be partitioned appropriately into

$$\begin{pmatrix} D_4 & U_4 V_5^* \\ U_5 V_4^* & D_5 \end{pmatrix} \left(\begin{array}{c|c} z_4 & Y_2^{top} \\ z_5 & Y_2^{bot} \end{array}\right) = \left(\begin{array}{c|c} b_4 & U_2^{top} \\ b_5 & U_2^{bot} \end{array}\right)$$
$$\textit{and} \quad \begin{pmatrix} D_6 & U_6 V_7^* \\ U_7 V_6^* & D_7 \end{pmatrix} \left(\begin{array}{c|c} z_6 & Y_3^{top} \\ z_7 & Y_3^{bot} \end{array}\right) = \left(\begin{array}{c|c} b_6 & U_3^{top} \\ b_7 & U_3^{bot} \end{array}\right),$$

where $b_4 = b(\mathcal{I}_4)$, $b_5 = b(\mathcal{I}_5)$, $b_6 = b(\mathcal{I}_6)$, and $b_7 = b(\mathcal{I}_7)$ are subvectors in the right-hand side b (notice that $\mathcal{I}_4 \cup \mathcal{I}_5 = \mathcal{I}_2$ and $\mathcal{I}_6 \cup \mathcal{I}_7 = \mathcal{I}_3$ according to the cluster tree in Fig. 1).

$$U_{\text{big}} = \left[\begin{array}{c|c} U_2 & U_4 \\ & U_5 \\ \hline U_3 & U_6 \\ & U_7 \end{array}\right] \qquad V_{\text{big}} = \left[\begin{array}{c|c} V_2 & V_4 \\ & V_5 \\ \hline V_3 & V_6 \\ & V_7 \end{array}\right]$$

Fig. 3: Concatenation of all low-rank bases into two big matrices for the HODLR matrix in Fig. 2 assuming the ranks at the same level are the same.

$$D_{\text{big}}^2 = \left[\begin{array}{c} D_4 \\ \hline D_5 \\ \hline D_6 \\ \hline D_7 \end{array}\right] \qquad K_{\text{big}}^1 = \left[\begin{array}{c} K_2 \\ \hline K_3 \end{array}\right] \qquad K_{\text{big}}^0 = \boxed{K_1}$$

Fig. 4: Concatenation of diagonal blocks and K matrices defined at (11) for the HODLR matrix in Fig. 2 assuming the ranks at the same level are the same. A superscript ℓ means a block-row view of the matrix, where matrix rows are partitioned according to nodes at the ℓ-th level in the tree.

At the second step, we have four subproblems:

$$\begin{cases} D_4\,[\, z_4 \,|\, Y_2^{top} \,|\, Y_4 \,] = [\, b_4 \,|\, U_2^{top} \,|\, U_4 \,] \\ D_5\,[\, z_5 \,|\, Y_2^{bot} \,|\, Y_5 \,] = [\, b_5 \,|\, U_2^{bot} \,|\, U_5 \,] \end{cases}$$
$$\textit{and} \quad \begin{cases} D_6\,[\, z_6 \,|\, Y_3^{top} \,|\, Y_6 \,] = [\, b_6 \,|\, U_3^{top} \,|\, U_6 \,] \\ D_7\,[\, z_7 \,|\, Y_3^{bot} \,|\, Y_7 \,] = [\, b_7 \,|\, U_3^{bot} \,|\, U_7 \,] \end{cases},$$

where the four D matrices are corresponding diagonal blocks (that have full ranks), and we solve them directly using, e.g., the LU factorization.

Observe the right-hand sides in the above four subproblems, where the U matrices have been partitioned and then concatenated together. This motivates the idea of concatenating all the U bases into one big matrix U_{big} as shown in Fig. 3. The same strategy is applied to form a big matrix V_{big} of all the V bases for reasons that will be clear when we introduce our GPU algorithms. We also concatenate diagonal blocks and K matrices defined at (11) *for ease of presentation.*

In general, we traverse $\mathcal{T}_L$ in a breadth-first (top-down level-by-level) order to create the two big matrices. Consider every node γ that has two children α and β (so α and β are siblings). First, we place U_α and U_β vertically on top of each other as $\begin{pmatrix} U_\alpha \\ U_\beta \end{pmatrix}$ (the order does not matter). In general, U_α and U_β may not have the same number of columns, and we align them to the left. We do the same thing with V_α and V_β. Second, if γ is not the root of $\mathcal{T}_L$, it is associated with low-rank bases U_γ

and V_γ, and we form the following concatenation

$$\left[\begin{array}{c|c} U_\gamma & \begin{matrix} U_\alpha \\ U_\beta \end{matrix} \end{array}\right] \quad \text{and} \quad \left[\begin{array}{c|c} V_\gamma & \begin{matrix} V_\alpha \\ V_\beta \end{matrix} \end{array}\right]. \tag{12}$$

Notice that the number of rows in U_γ equals to the sum of those in U_α and U_β and the same applies to the V matrices. Finally, we obtain two big matrices U_{big} and V_{big} after traversing $\mathcal{T}_L$. With the previous assumption of constant rank, U_{big} and V_{big} are two matrices of size N-by-rL. In general, when ranks of off-diagonal blocks vary, U_{big} and V_{big} are no longer rectangular matrices. But the algorithms we are going to introduce can be generalized accordingly. In the same spirit, we concatenate the D matrices and K matrices as shown in Fig. 4.

Given a cluster tree $\mathcal{T}_L$ as defined in Definition 1, we can unroll the previous recursive algorithm into a for-loop based algorithm, which consists of a factorization stage and a solution stage as described in Algorithms 1 and 2, respectively. The advantages of the new data structure are clear. We need only one BLAS or LAPACK call for the computation involving multiple left low-rank bases across different levels in $\mathcal{T}_L$, and there is no unnecessary data movement. For example, one LAPACK call (getrs) is sufficient for solving all the right-hand sides at a leaf node in $\mathcal{T}_L$ (line 4 in Algorithm 1).

Notice that a lot of memory allocation in Algorithm 1 is avoided by overwriting inputs with outputs. For example, Y_{big} overwrites U_{big}, and the factorizations of diagonal blocks are stored in place. As a result, Algorithm 1 requires little extra memory when the rank r is small.

C. Parallel algorithms for GPUs

Recall that in the recursive algorithm, the two subproblems in (7) are independent of each other and can be solved in parallel. Consider the underlying cluster tree of a HODLR matrix. If we associate the root with the task of solving the original linear system and every other tree node with the task of solving a corresponding subproblem, then the cluster tree is effectively a *task graph*. In the task graph, every node depends on its two children if they exist. The same analysis applies to the factorization stage and the solution stage as described in Algorithms 1 and 2, respectively, and the underlying task graphs are exactly the same (definitions of tasks differ).

Let us focus on the factorization stage, and most of the following discussion applies to the solution stage as well. It is clear that in the underlying task graph, all the nodes (tasks) at the same level are embarrassingly parallel. In other words, the two for-loops at lines 2&7 in Algorithm 1 can be parallelized. Indeed, the HODLRLIB library [33] employs the "parallel-for" directive in OpenMP [44] to parallelize the two for-loops. In Algorithm 1, it is also obvious that the nodes (tasks) at the same level work on different subblocks in U_{big} and V_{big}, which correspond to a (consecutive) partitioning of the row indices.

Notice that in the factorization stage, every leaf task requires solving a linear system with multiple right-hand sides and every nonleaf task comprises solving a linear system and matrix-matrix multiplications (gemm's). Therefore, Algorithm 1 can

Algorithm 1 Nonrecursive algorithm for solving (4): factorization stage

Input: diagonal blocks in A, matrix U_{big}, matrix V_{big}, and cluster tree $\mathcal{T}_L$.
Output: matrix Y_{big} and stored factorizations.

1: $Y_{\text{big}} \leftarrow U_{\text{big}}$ *// Y_{big} overwrites U_{big}.*
2: **for** leaf node α in $\mathcal{T}_L$ **do**
3: Factorize diagonal block $D_\alpha = A(\mathcal{I}_\alpha, \mathcal{I}_\alpha)$ and store its LU factorization in-place.
4: Apply D_α^{-1} (the previous LU factorization) to solve multiple right-hand sides $Y_{\text{big}}(\mathcal{I}_\alpha, :)$ in-place.
5: **end for**
6: **for** level $\ell = L-1$ **to** 0 **do**
7: **for** tree node γ at level ℓ in $\mathcal{T}_L$ **do**
8: Let the children of γ be α and β.
9: Factorize $K_\gamma = \begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix}$ and store its LU factorization in-place.
10: Apply K_γ^{-1} (the previous LU factorization) to solve

$$\begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix} \begin{pmatrix} W_\alpha \\ W_\beta \end{pmatrix} = \begin{pmatrix} V_\alpha^* Y_{\text{big}}(\mathcal{I}_\alpha, 1:r\ell) \\ V_\beta^* Y_{\text{big}}(\mathcal{I}_\beta, 1:r\ell) \end{pmatrix}. \tag{13}$$

11: Compute

$$Y_{\text{big}}(\mathcal{I}_\gamma, 1:r\ell) \leftarrow Y_{\text{big}}(\mathcal{I}_\gamma, 1:r\ell) - \begin{pmatrix} Y_\alpha W_\alpha \\ Y_\beta W_\beta \end{pmatrix}. \tag{14}$$

12: **end for**
13: **end for**

Algorithm 2 Nonrecursive algorithm for solving (4): solution stage

Input: matrix Y_{big}, matrix V_{big}, stored LU factorizations, cluster tree $\mathcal{T}_L$, and right-hand side b.
Output: solution of (4), namely, x.

1: $x \leftarrow b$ *// x overwrites b.*
2: **for** leaf node α in $\mathcal{T}_L$ **do**
3: Apply D_α^{-1} (precomputed LU factorization) to solve right-hand side $x(\mathcal{I}_\alpha)$ in-place.
4: **end for**
5: **for** level $\ell = L-1$ **to** 0 **do**
6: **for** tree node γ at level ℓ in $\mathcal{T}_L$ **do**
7: Let the children of γ be α and β.
8: Apply K_γ^{-1} (precomputed LU factorization) to solve

$$\begin{pmatrix} V_\alpha^* Y_\alpha & I \\ I & V_\beta^* Y_\beta \end{pmatrix} \begin{pmatrix} w_\alpha \\ w_\beta \end{pmatrix} = \begin{pmatrix} V_\alpha^* x(\mathcal{I}_\alpha) \\ V_\beta^* x(\mathcal{I}_\beta) \end{pmatrix}. \tag{15}$$

9: Compute

$$x(\mathcal{I}_\gamma) \leftarrow x(\mathcal{I}_\gamma) - \begin{pmatrix} Y_\alpha w_\alpha \\ Y_\beta w_\beta \end{pmatrix}. \tag{16}$$

10: **end for**
11: **end for**

TABLE I: Notations for Algorithms 3 and 4

superscript ℓ	block-row view of a matrix, where matrix rows are partitioned according to nodes at the ℓ-th level in $\mathcal{T}_L$.
operator $\odot$	block-wise matrix multiplication between two block-row matrices (the result is a block-row matrix).
$Y^{\ell+1}$	shorthand for $Y_{\text{big}}^{\ell+1}(:, r\ell+1 : r(\ell+1))$.
$V^{\ell+1}$	shorthand for $V_{\text{big}}^{\ell+1}(:, r\ell+1 : r(\ell+1))$.
$\left(V^{\ell+1}\right)^*$	block-wise transpose of $V^{\ell+1}$.
$\left(D_{\text{big}}^L\right)^{-1}$	applying matrix inverse (solution of linear systems) with block-wise LU factorizations.
$\left(K_{\text{big}}^{\ell}\right)^{-1}$	

be transformed into a parallel algorithm by simply batching the BLAS and LAPACK calls from tasks at the same level. In particular, we take advantage of the batched LU factorization/solution (getrfBatched/getrsBatched) and the batched gemm (gemmBatched) from cuBLAS[1] in our GPU algorithms. With our new data structure that concatenates the left low-rank bases into one (big) matrix, we need only one cuBLAS call for computations involving multiple left low-rank bases across different levels in the tree. This property simplifies the implementation, avoids unnecessary data movement, and reduces the number of kernel launches. We present the pseudocode of our GPU factorization algorithm in Algorithm 3.

The new data structure also allows us to take advantage of an optimization of the general batched gemm kernel—gemmStridedBatched[2]—from cuBLAS when the (left) low-rank bases at the same level have the same sizes. In other words, there is a constant stride among the corresponding subblocks in U_{big} and V_{big} that are accessed by a batched gemm kernel. Such an optimization improves the performance significantly when the sizes of input matrices are small.

The solution stage as described in Algorithm 2 has similar structure as the factorization stage. It is straightforward to see that the two for-loops at lines 2&6 in Algorithm 2 can be parallelized. Again, we batch the BLAS and LAPACK calls from tasks at the same level. With the precomputed data from the factorization stage, the solution stage is relatively simple as described in Algorithm 4.

Notice that for the first few levels the number of nodes is small, and we empirically found that the batched gemm kernel was outperformed by launching independent gemm kernels using CUDA streams. Before ending this section, we point out two variants of (9):

$$\begin{pmatrix} I & V_\beta^* Y_\beta \\ V_\alpha^* Y_\alpha & I \end{pmatrix} \begin{pmatrix} w_\alpha \\ w_\beta \end{pmatrix} = \begin{pmatrix} V_\beta^* z_\beta \\ V_\alpha^* z_\alpha \end{pmatrix}$$

and

$$\begin{pmatrix} I & V_\alpha^* Y_\alpha \\ V_\beta^* Y_\beta & I \end{pmatrix} \begin{pmatrix} w_\beta \\ w_\alpha \end{pmatrix} = \begin{pmatrix} V_\alpha^* z_\alpha \\ V_\beta^* z_\beta \end{pmatrix},$$

[1]https://docs.nvidia.com/cuda/cublas/index.html

[2]https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply/

Algorithm 3 GPU algorithm for solving (4): factorization stage (see notations in Table I)

Input: matrix D_{big}, matrix U_{big}, matrix V_{big}, and cluster tree $\mathcal{T}_L$.

Output: matrix Y_{big}, block-wise factorizations of matrices D_{big}^L, K_{big}^0, K_{big}^1, …, K_{big}^{L-1}.

1: $Y_{\text{big}} \leftarrow U_{\text{big}}$ *// Y_{big} overwrites U_{big}.*

2: BATCHED_LU_FACTORIZE_INPLACE(D_{big}^L)

3: BATCHED_LU_SOLVE_INPLACE:

$$Y_{\text{big}}^L \leftarrow \left(D_{\text{big}}^L\right)^{-1} \odot Y_{\text{big}}^L$$

4: **for** level $\ell = L-1$ **to** 0 **do**

5: BATCHED_GEMM:

$$T^{\ell+1} \leftarrow \left(V^{\ell+1}\right)^* \odot Y^{\ell+1}$$

// T and W (below) are temporary variables; see (13).

6: BATCHED_GEMM:

$$W^{\ell+1} \leftarrow \left(V^{\ell+1}\right)^* \odot Y_{\text{big}}^{\ell+1}(:, 1 : r\ell)$$

7: Form matrix K_{big}^{ℓ} with $T^{\ell+1}$. *// See* (11) *and Fig. 4.*

8: BATCHED_LU_FACTORIZE_INPLACE(K_{big}^{ℓ})

9: BATCHED_LU_SOLVE_INPLACE:

$$W^{\ell} \leftarrow (K_{\text{big}}^{\ell})^{-1} \odot W^{\ell}$$

10: BATCHED_GEMM:

$$Y_{\text{big}}^{\ell+1}(:, 1 : r\ell) \leftarrow Y_{\text{big}}^{\ell+1}(:, 1 : r\ell) - Y^{\ell+1} \odot W^{\ell+1}$$

11: **end for**

Algorithm 4 GPU algorithm for solving (4): solution stage (see notations in Table I)

Input: matrix Y_{big}, matrix V_{big}, (block-wise factorizations of) matrices D_{big}^L, K_{big}^0, K_{big}^1, …, K_{big}^{L-1}, cluster tree $\mathcal{T}_L$, and right-hand side b.

Output: solution x.

1: $x \leftarrow b$ *// x overwrites b.*

2: BATCHED_LU_SOLVE_INPLACE:

$$x^L \leftarrow \left(D_{\text{big}}^L\right)^{-1} \odot x^L$$

3: **for** level $\ell = L-1$ **to** 0 **do**

4: BATCHED_GEMM:

$$w^{\ell+1} \leftarrow \left(V^{\ell+1}\right)^* \odot x^{\ell+1}$$

// w is a temporary variable; see (15).

5: BATCHED_LU_SOLVE_INPLACE:

$$w^{\ell} \leftarrow (K_{\text{big}}^{\ell})^{-1} \odot w^{\ell}$$

6: BATCHED_GEMM:

$$x^{\ell+1} \leftarrow x^{\ell+1} - Y^{\ell+1} \odot w^{\ell+1}$$

7: **end for**

where the coefficient matrices have identities on the diagonals but either the right-hand side or the solution is reordered. Although (9) is mathematically equivalent to the two alternatives, they may perform differently on a GPU. To be concrete, we typically need partial pivoting for the (batched) LU factorization (line 7 in Algorithm 3) with the formulation in (9), which impedes performance. The alternative formulations could circumvent this issue, but they require an extra cost of shuffling the right-hand side or the solution.

D. Complexity analysis

Given a rank-r HODLR matrix $A \in \mathbb{R}^{N\times N}$ with the underlying cluster tree $\mathcal{T}_L$, we assume the off-diagonal blocks in A have the same rank r and the diagonal blocks in A have the same size $m = N/2^L$. In practice, we usually prescribe m to be a small constant independent of N, or equivalently we set $L = \mathcal{O}(\log(N))$. The following analysis and results can be easily extended to general cases where the ranks and sizes of diagonal blocks vary.

Let us first consider the storage of A that includes

- Diagonal blocks: $m^2 \times 2^L = mN$.
- Off-diagonal blocks (U_{big} and V_{big}): $2 \times N \times r \times L = 2rNL$.

As explained earlier, the factorization of the HODLR matrix can be done in-place, which requires little extra memory. We summarize the above as the following theorem.

Theorem 2 (Storage)**.** *The storage of the HODLR matrix A and its factorization is*

$$m_f = mN + rNL = \mathcal{O}(rN\log(N)).$$

Next, we consider the computational cost for factorizing matrix A and summarize the results in a theorem.

- Leaf level (lines 3&4 in Algorithm 1):

  $$\frac{2}{3}m^3 \times 2^L + 2m^2 \times r \times L \times 2^L = \frac{2}{3}m^2N + 2mrNL,$$

  where factorizing an $m \times m$ matrix using the LU factorization requires $2/3m^3$ operations and solving one right-hand side using the LU factorization requires $2m^2$ operations with forward and backward substitutions.
- At level $\ell\,(1 \le \ell \le L)$, we solve (13) and compute (14). Assuming the rank r is small, the cost is dominated by conducting matrix-matrix multiplication[3] to form the right-hand sides in (13) and the update in (14), which is

  $$2r^2N\ell + 2r^2N\ell = 4r^2N\ell$$

  operations. Notice this cost is linear with respect to ℓ.

[3]The computational cost for multiplying an $n\times k$ (real) matrix and a $k\times m$ (real) matrix is $2knm$ operations.

Theorem 3 (Factorization cost)**.** *The factorization of the HODLR matrix A requires the following amount of operations:*

$$\begin{aligned} t_f &= \frac{2}{3}m^2N + 2mrNL + \sum_{\ell=1}^{L} 4r^2N\ell \\ &= \frac{2}{3}m^2N + 2mrNL + 2r^2N(L+L^2) \\ &= \mathcal{O}(r^2N\log^2(N)). \end{aligned}$$

Finally, we consider the computational cost for solving an arbitrary right-hand side using the factorization of matrix A and summarize the results in a theorem.

- Leaf level (line 3 in Algorithm 2):

  $$2m^2 \times 2^L = 2mN,$$

  where 2^L is the number of leaf nodes in the cluster tree (diagonal blocks in A).
- At level $\ell\,(1 \le \ell \le L)$, we solve (15) and compute (16). Assuming the rank r is small, the cost is dominated by conducting matrix-vector multiplication to form the right-hand side in (15) and the update in (16), which is

  $$2rN + 2rN = 4rN$$

  operations. Notice that this cost does not depend on ℓ.

Theorem 4 (Solution cost)**.** *The solution of an arbitrary right-hand side using the factorization of A requires the following amount of operations:*

$$t_s = 2mN + \sum_{\ell=1}^{L} 4rN = 2mN + 4rNL = \mathcal{O}(rN\log(N)). \tag{17}$$

It is not a coincidence that the amount of required operations in theorem 4 is twice as many as the amount of storage in theorem 2. The fact is that every stored entry is touched once and is involved with two operations (a multiplication and an addition) in the solution stage.

Remark 1 (Numerical rank)**.** *From users' perspective, it is usually more desirable to specify a tolerance for low-rank compressions rather than the ranks directly. Consider using HODLR approximations for problems that are not highly oscillatory. When the underlying problem lies in 1D, the ranks of all off-diagonal blocks are roughly the same and are independent of the problem size for a prescribed tolerance. As the above analysis shows, the factorization and the solution both scale nearly linearly. However, when the underlying problem is in 2D or 3D, the ranks increase with the problem size, and the asymptotic complexities deteriorate (see, e.g., Remark 5.1 in [7]).*

E. Two perspectives and connections

The HODLR format is relatively simple among other formats that approximate off-diagonal blocks of a matrix, but the described algorithms have close connections to algorithms for more complicated formats. Below, let us review two other perspectives on factorizing a HODLR matrix and discuss their connections to related algorithms.

a) Matrix factorization: The first perspective is from a matrix factorization point of view, which initially appeared in [1]. Let us illustrate this with the following example. (An interesting extension is the computation of a symmetric factorization of a HODLR matrix that is SPD [7], which has various applications involving covariance matrices.)

Example 2. *Consider the HODLR matrix A in Fig. 2, where the underlying cluster tree has three levels (level 0, 1, and 2). Suppose we apply Algorithm 1 to A. Equivalently, we obtain the following factorization (and its "inverse")*

$$A = \begin{pmatrix} A_2 & U_2V_3^* \\ U_3V_2^* & A_3 \end{pmatrix} = \begin{pmatrix} A_2 & \\ & A_3 \end{pmatrix} \begin{pmatrix} I & Y_2V_3^* \\ Y_3V_2^* & I \end{pmatrix}$$
$$= \underbrace{\begin{pmatrix} A_4 & & & \\ & A_5 & & \\ & & A_6 & \\ & & & A_7 \end{pmatrix}}_{A^{(3)}} \underbrace{\begin{pmatrix} I & Y_4V_5^* & & \\ Y_5V_4^* & I & & \\ & & I & Y_6V_7^* \\ & & Y_7V_6^* & I \end{pmatrix}}_{A^{(2)}}$$
$$\underbrace{\begin{pmatrix} I & Y_2V_3^* \\ Y_3V_2^* & I \end{pmatrix}}_{A^{(1)}}.$$

Notice that the Y matrices are among the outputs of Algorithm 1. In addition, we can apply A^{-1} easily because we can apply the inverses of $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ easily. For example, we have stored the factorizations of every diagonal block in $A^{(3)}$ (line 3 in Algorithm 1), and every 2×2 diagonal block in $A^{(2)}$ can be inverted according to (10)*, where the factorization of a small matrix has been stored (line 9 in Algorithm 1). It should not be surprising that the algorithm of applying A^{-1} to a vector b is just Algorithm 2. We summarize the discussion into the following:*

Theorem 5. *Algorithm 1 and Algorithm 2 are equivalent to computing a matrix factorization of a HODLR matrix and applying the inverse of the factorization to a vector, respectively.*

Notice that the above matrix factorization of A naturally leads to an efficient algorithm for evaluating the determinant of A. Observe that

$$det(A) = det(A^{(1)})det(A^{(2)})det(A^{(3)}),$$

where $det(A^{(3)})$ can be readily obtained with the LU factorizations of its diagonal blocks, the determinant of every 2×2 diagonal block in $A^{(2)}$ and $A^{(1)}$ can be evaluated efficiently with the stored factorizations according to Sylvester's determinant theorem [8].

b) Extended sparse linear system: The other perspective is related to sparse matrix algebra, where we embed a HODLR matrix into a larger sparse matrix [2]. To be more specific, the original dense problem (4) is equivalent to a larger sparse linear system, and we solve the sparse problem with straightforward Gaussian elimination to obtain the solution of the original problem. The power of this approach is that it also generalizes to solving problems involving more complicated formats of hierarchical low-rank compression. Let us illustrate the main idea through the following example.

Example 3. *Consider solving a dense linear system*

$$\begin{pmatrix} A_2 & U_2V_3^* \\ U_3V_2^* & A_3 \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}.$$

With two auxiliary variables $w_3 = V_2^*x_2$ and $w_2 = V_3^*x_3$ (the subscripts of w's are chosen for the convenience of the following presentation), we rewrite the above dense linear system as the following block-sparse linear system:

$$\begin{pmatrix} A_2 & & U_2 & \\ & A_3 & & U_3 \\ V_2^* & & & -I \\ & V_3^* & -I & \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \\ 0 \\ 0 \end{pmatrix}.$$

The beauty is that we can solve this sparse linear system with Gaussian elimination in a straightforward manner: we factorize the block-sparse matrix with the block-LU factorization and then solve with block-forward and block-backward substitutions. To be concrete, we first apply two steps of block Gaussian elimination (for x_2 and x_3) and obtain the resulting linear system involving the resulting Schur complement

$$\begin{pmatrix} V_2^*Y_2 & I \\ I & V_3^*Y_3 \end{pmatrix} \begin{pmatrix} w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} V_2^*z_2 \\ V_3^*z_3 \end{pmatrix},$$

where $Y_2 = (A_2)^{-1}U_2$, $Y_3 = (A_3)^{-1}U_3$, $z_2 = (A_2)^{-1}b_2$, and $z_3 = (A_3)^{-1}b_3$. Readers may recognize this small linear system as (9)*. After solving the above system, we get the original solution via forward and backward substitutions, which turns out to be exactly the same as* (8)*:*

$$\begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} z_2 \\ z_3 \end{pmatrix} - \begin{pmatrix} Y_2w_2 \\ Y_3w_3 \end{pmatrix}.$$

This approach extends to solving linear systems involving HSS matrices[4]. It is shown in [5] (Section 7.2.3) that applying Gaussian elimination to an extended sparse linear system introduces *no* fill-in with an appropriate ordering. Alternatively, we can feed an extended sparse linear system directly to a highly optimized sparse direct solver as proposed by Ho and Greengard [45] for computational efficiency and stability.

Both the HODLR and the HSS formats rely on the so-called *weak admissibility*, and related algorithms are relatively simple. By contrast, the compression format associated with the FMM employs the *strong admissibility*, and solving a linear system involving such a matrix is still an active research question. Recent advances [28], [46], [47] demonstrate that the approach of creating an extended sparse linear system is still viable as long as we compress fill-in blocks during Gaussian elimination. It turns out that this improvement can also be applied to solve large *sparse* linear systems arising from the discretization of linear elliptic partial differential equations that are not highly indefinite [48]–[51].

[4]If the low-rank bases in a HODLR matrix are *nested* (the bases of a nonleaf node can be constructed from those of its two children), then the HODLR matrix is an HSS matrix.

TABLE II: Notations for numerical results

N	problem size/matrix size/degrees of freedom/number of points
t_f	factorization time in seconds
t_s	solution time for a random right-hand side in seconds
mem	memory of the factorization in gigabytes (GB)
relres	relative residual of a solution $\tilde{x}$, i.e., $\|b - A\tilde{x}\|/\|b\|$

IV. Numerical Results

We benchmark the speed and the accuracy of our GPU solver and compare it against existing methods in the literature. In section IV-A, we apply our GPU solver to kernel matrices and compare to HODLRLIB [33]. In sections IV-B and IV-C, we apply our HODLR solver to linear systems coming from the discretization of BIEs with a Laplace double-layer potential and a Helmholtz potential, respectively. We compare to the block-sparse solver proposed by Ho and Greengard [45], which can leverage state-of-the-art sparse direct solvers. The machine where our numerical experiments were performed has

- two Intel Xeon Gold 6254 CPUs, each with 18 cores at 3.10 GHz (peak performance $\approx$ 1.27 TFlop/s),
- an NVIDIA GPU that is a Tesla V100 GPU with 32 GB of memory (peak performance $\approx$ 7 TFlop/s).
- a PCIe 3.0 $\times$16 between the CPU and the GPU that can deliver up to 15.75 GB/s.

Our GPU code was compiled with the NVIDIA compiler nvcc (version 11.3.58) and linked with the cuBLAS library (version 11.4.2.10064) on the Linux OS (5.4.0-72-generic.x86_64). Calculations are performed with double-precision floating-point arithmetic unless stated otherwise. Notations that we use to report results are shown in Table II, and the timings were taken as the average of five consecutive runs.

A. Kernel matrix

Consider solving (1), where matrix A is generated from the Rotne-Prager-Yamakawa (RPY) tensor kernel [52], [53], which is frequently used to model the hydrodynamic interactions in simulations of Brownian dynamics. Given a set of points $\{y_i\}_{i=1}^N$, the kernel matrix A is defined as

$$A_{i,j} = \begin{cases} \frac{kT}{8\pi\eta|r|}\left[I + \frac{r\bigotimes r}{|r|^2} + \frac{2a^2}{3|r|^2}(I - 3\frac{r\bigotimes r}{|r|^2})\right] & \text{if } r \geq 2a, \\ \frac{kT}{6\pi\eta a}\left[(1 - \frac{9}{32}\frac{|r|}{a})I + \frac{3}{32a}\frac{r\bigotimes r}{|r|}\right] & \text{if } r < 2a, \end{cases} \quad (18)$$

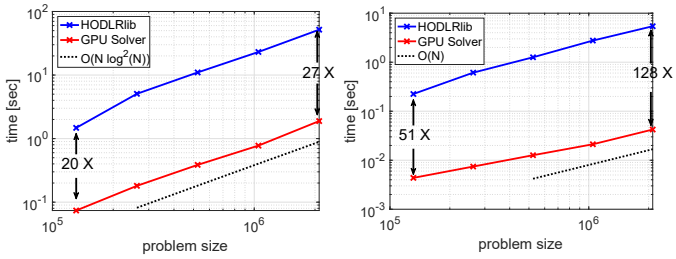
where $r = y_i - y_j$ and $|r|$ denotes its two norm.

We approximated matrix A with a HODLR matrix and obtained an approximate solution for (1). In the following, we compare our GPU solver to the HODLRLIB library [33], which is a C++ code parallelized with OpenMP for shared-memory multicore architectures. To be consistent with the benchmark of HODLRLIB[5], we randomly generated $\{x_i\}_{i=1}^N$ with a uniform distribution over $[-1, 1]$, and set $k = T = \eta = 1$, $a = |r|_{\min}/2$ with $|r|_{\min}$ being the minimum distance among all pairs of x_i and x_j. We compiled HODLRLIB with g++ 7.5.0 and linked it with the (sequential) Intel MKL library of version 20.0. Timing results of HODLRLIB were obtained on two Intel Xeon CPUs of 36 cores combined.

[5]https://hodlrlib.readthedocs.io/en/latest/benchmarks.html

TABLE III: *Results for solving* (1) *with the RPY kernel defined in* (18)*. The low-rank approximation accuracy is prescribed to be* 10^{-12}.

N	HODLRLIB t_f	HODLRLIB t_s	GPU Solver t_f	GPU Solver t_s	mem	relres
$2^{17} = 131,072$	1.47	0.22	7.39e-2	4.37e-3	0.88	1.68e-11
$2^{18} = 262,144$	5.09	0.61	1.81e-1	7.43e-3	1.93	2.57e-9
$2^{19} = 524,288$	10.9	1.26	3.86e-1	1.27e-2	4.23	5.28e-11
$2^{20} = 1,048,576$	23.1	2.76	7.75e-1	2.12e-2	8.94	1.32e-9
$2^{21} = 2,097,152$	51.7	5.42	1.89e-0	4.23e-2	19.2	1.10e-9



(a) Factorization time (b) Solution time

Fig. 5: Comparison between (1) the HODLRLIB library [33] running on two Intel Xeon Gold 6254 18-core CPUs and (2) our GPU solver running on an NVIDIA Tesla V100 GPU.

We constructed HODLR approximations on the CPUs, where diagonal blocks have size 64×64 and off-diagonal blocks are compressed using the LowRank::rookPiv() function, an approximate partial-pivoted LU, in HODLRLIB. For our GPU solver, we copied data including D_{big}, U_{big}, and V_{big} to the GPU, where we typically used a significant portion of the bandwidth, around 12 GB/s, in our experiments.

With a prescribed accuracy of 10^{-12} for low-rank compression, we obtained the results in Table III, where the factorization time and the solution time are plotted in Fig. 5. From these results, we make two observations. First, the factorization time, the solution time, and the memory footprint all scale nearly linearly as the theory in section III-D predict. The increase of the solution time is slower than $N\log(N)$ as given in (17) and is close to being linear. The reason is that the GPU utilization increases as the problem size N increases. Second, our GPU solver achieved significant speedups against HODLRLIB. In particular, the speedup becomes larger as the problem size increases, and the acceleration of the solution time is larger than that of the factorization time. As mentioned earlier, the parallelization in HODLRLIB is only across nodes at the same level (no parallelization within a tree node), while our GPU solver also employs parallelization inside every tree node. For $N = 2^{21}$, the factorization and the solution of our GPU solver achieved 878 Gflop/s and 119 Gflop/s, respectively.

TABLE IV: *Results for solving* (21) *discretized with a 2nd-order quadrature. In (b), all calculations were performed in single precision except for the sequential block-sparse solver (Matlab backslash for sparse matrices does not support single precision).*

N	Serial HODLR Solver			Serial Block-Sparse Solver			Parallel Block-Sparse Solver			GPU HODLR Solver			relres
	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	
$2^{18} = 262,144$	4.51e+1	5.93e-1	1.09	2.87e+0	1.33e-1	0.57	7.03e+0	1.85e-2	3.56	6.94e-2	4.87e-3	1.09	2.10e-9
$2^{19} = 524,288$	9.73e+1	1.05e-0	2.25	5.88e+0	2.86e-1	1.14	1.37e+1	3.74e-2	7.08	1.40e-1	8.19e-3	2.25	7.13e-9
$2^{20} = 1,048,576$	2.20e+2	2.18e-0	4.63	1.21e+1	5.09e-1	2.28	2.89e+1	8.30e-2	14.2	2.90e-1	1.28e-2	4.63	5.60e-9
$2^{21} = 2,097,152$	4.76e+2	4.99e-0	9.46	2.35e+1	1.00e-0	4.56	6.20e+1	1.82e-1	28.6	6.10e-1	2.40e-2	9.46	7.82e-9
$2^{22} = 4,194,304$	1.05e+2	9.81e-0	19.3	4.90e+1	2.29e-0	9.15	1.29e+2	5.18e-1	56.9	1.25e-0	4.61e-2	19.3	1.31e-8

(a) *High-accuracy solver*

N	Serial HODLR Solver			Serial Block-Sparse Solver			Parallel Block-Sparse Solver			GPU HODLR Solver			relres
	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	
$2^{18} = 262,144$	1.41e+0	3.07e-1	0.27	1.66e+0	8.44e-2	0.35	4.50e+1	9.80e-3	2.40	1.74e-2	2.66e-3	0.27	3.13e-5
$2^{19} = 524,288$	2.95e+0	5.82e-1	0.55	3.32e+0	1.68e-1	0.69	9.42e+1	1.77e-2	4.79	3.39e-2	3.92e-3	0.55	1.49e-4
$2^{20} = 1,048,576$	5.88e+0	1.16e-0	1.09	6.55e+0	3.42e-1	1.39	1.99e+1	3.71e-2	9.56	5.79e-2	6.48e-3	1.09	7.20e-5
$2^{21} = 2,097,152$	1.21e+1	2.48e-0	2.13	1.32e+1	6.89e-1	2.77	3.86e+1	7.83e-2	19.1	1.29e-1	1.09e-2	2.13	6.11e-4
$2^{22} = 4,194,304$	2.47e+1	5.40e-0	4.26	2.79e+1	1.36e-0	5.55	8.49e+1	2.04e-1	38.3	2.70e-1	2.05e-2	4.26	2.07e-4
$2^{23} = 8,388,608$	5.03e+1	1.08e+1	8.45	5.81e+1	2.87e-0	11.1	1.73e+2	3.99e-1	76.2	4.26e-1	4.06e-2	8.45	4.04e-4
$2^{24} = 16,777,216$	1.19e+2	1.93e+1	17.0	1.18e+2	6.23e-0	23.1	3.77e+2	7.83e-1	152	8.58e-1	8.38e-2	17.0	7.12e-4

(b) *Low-accuracy solver*

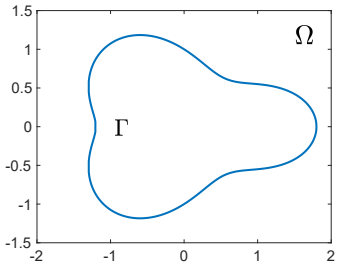


Fig. 6: A smooth contour Γ on the plane. The problem domain Ω, where PDEs (19) and (22) are defined, is *exterior* to Γ.

B. Laplace equation on an exterior domain

We next consider a boundary integral equation obtained by rewriting the BVP

$$\begin{cases} -\Delta u(x) = 0 & \text{in } \Omega, \\ u(x) = f(x) & \text{on } \Gamma, \end{cases} \tag{19}$$

where Ω is the infinite domain that is exterior to the smooth contour Γ shown in Fig. 6. To ensure a unique solution that is physically meaningful, we also require that there exists a real number $Q \in \mathbb{R}$ such that

$$\lim_{|x|\to\infty} \left(u(x) + \frac{Q}{2\pi} \log|x| \right) = 0. \tag{20}$$

We reformulate (19) as

$$\frac{1}{2}\sigma(x) + \int_\Gamma \left(d(x,y) - \frac{1}{2\pi} \log|x-z| \right) \sigma(x) ds(y) = f(x), \tag{21}$$

where

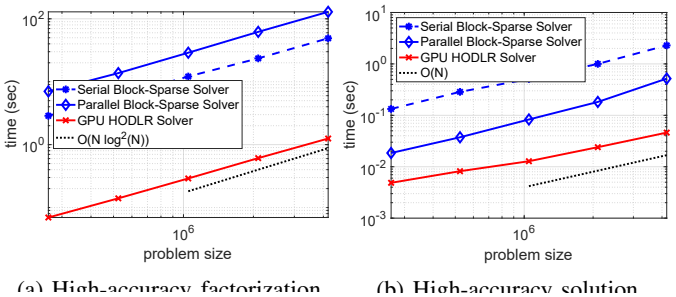
$$d(x,y) = \frac{n(y)\cdot(x-y)}{2\pi|x-y|^2}$$

is the "double layer kernel" associated with the Laplace equation, where $n(y)$ is a normal vector at a point $y \in \Gamma$, and where z is a fixed point in the interior of Γ [54], [55]. We chose z to be the origin and discretized (19) using the Trapezoidal rule. To solve the resulting linear system, we constructed a HODLR approximation using the proxy surface technique (see, e.g., [9, Chapter 17]) on a CPU and copied the HODLR matrix to the GPU.

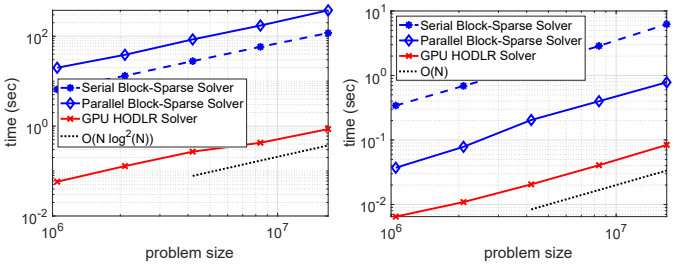
For comparison, we implemented the block-sparse solver by Ho and Greengard [45]. The block-sparse solver builds an HSS approximation [56] in a block sparse format and employs a sparse direct solver to solve the block sparse matrix. This approach takes advantage of the stability and the efficiency of state-of-the-art sparse direct solvers. We refer interested readers to [45] for comparisons between the block-sparse solver and an iterative solver employing the fast multipole method. For our problems, we empirically found that the Matlab backslash (calling UMFPACK [57], [58]) and the MKL PARDISO solver [59], [60] performed best on a single core and on all 36 cores of two Intel Xeon CPUs, respectively[6]. We refer to them as the sequential and the parallel block-sparse solvers.

Table IV shows the comparison between the block-sparse solver and our GPU solver, where the factorization time and the solution time are plotted in Fig. 7. In practice, it is not necessary to use as many discretization points for (21) on the smooth contour Γ. Here, our purpose is to benchmark the speed and the accuracy of our GPU solver. From the results, we make the following observations. First, the scaling of the factorization, the solution time, and the memory footprint of our GPU solver increase nearly linearly with the problem size.

[6]For UMFPACK, we used the [L,U,p] = lu(A,'vector') routine in Matlab, which applied only row-permutation on the input sparse matrix and avoided the overhead of computing a column permutation. For parallel performance on the two Intel Xeon Gold 6254 CPUs, we compared several popular sparse direct solvers for our matrices, and our conclusion is consistent with observations in the literature (see, e.g., [61]).



(a) High-accuracy factorization (b) High-accuracy solution



(c) Low-accuracy factorization (d) Low-accuracy solution

Fig. 7: Comparison between the block-sparse solver [45] and our GPU solver for solving (21) discretized with a 2nd-order quadrature.

Second, our GPU solver achieved significant speedups compared to the block-sparse solver, as shown in Fig. 7. Although the sequential HODLR solver is slower than the sequential block-sparse solver, it is well understood that parallelizing a sparse direct solver, which the block-sparse solver relies on, is challenging [62]. In fact, the parallel factorization stage, which consists of a symbolic factorization and a numerical factorization, is slower than the sequential counterpart. What happened here was that we did not compute any fill-in reducing ordering in the sequential method (the block-sparse matrix has a special structure that natural ordering turns out working well), but the parallel method spent a lot of time (and storage) in symbolic factorization to find parallelism (so the subsequent numerical factorization and the solution can be accelerated). Compared to the sequential method, the parallel block-sparse solver also consumed more memory from the analysis in the symbolic factorization phase that the solver needs in the numerical factorization and solve phases.

Third, one advantage of our method is that the accuracy is tunable. With highly accurate low-rank compressions, we obtained fast direct solvers as shown in Table IVa. With reasonably accurate low-rank compressions, we obtained results in Table IVb, where we were able to take advantage of single-precision floating-point arithmetic. The use of single-precision calculations accelerated the factorization and the solution by approximately $2\times$, and it also reduced the memory footprint by $2\times$.

C. Helmholtz equation on an exterior domain

Consider the following exterior BVP that models, e.g., time-harmonic wave problems,

$$\begin{cases} -\Delta u(x) - \kappa^2 u(x) = 0 & \text{in } \Omega, \\ u(x) = f(x) & \text{on } \Gamma. \end{cases} \tag{22}$$

Observe that (22) is defined on an *infinite* domain, as shown in Fig. 6. To ensure a well-posed problem, we also require the radiation condition at infinity: for every unit vector z

$$\lim_{r\to\infty} \sqrt{r}\left(\frac{\partial u(rz)}{\partial r} - i\kappa u(rz)\right) = 0. \tag{23}$$

Solving (22) by directly discretizing the PDE is quite challenging. Instead, we use a BIE formulation. Define the single- and double-layer kernels

$$s_\kappa(x,y) = \phi_\kappa(x-y),$$
$$d_\kappa(x,y) = n(y)\cdot\nabla_y\phi_\kappa(x-y),$$

where $\phi_\kappa(x) = \frac{i}{4}H_0^{(1)}(\kappa|x|)$ is the fundamental solution of the Helmholtz operator and $H_0^{(1)}$ is the zeroth-order Hankel function of the first kind, and where $n(y)$ is the inward normal at $y \in \Gamma$. We reformulate (22) as a second-kind Fredholm equation [54], [55]

$$\frac{1}{2}\sigma(x) + \int_\Gamma \left(d_\kappa(x,y) + i\eta s_\kappa(x,y)\right)\sigma(x)ds(y) = f(x), \tag{24}$$

where $x \in \Gamma$ and η is a parameter that is often chosen as $\eta = \pm\kappa$. The BIE is defined on a finite domain and automatically satisfies (20). We chose $\eta = \kappa = 100$ and discretized (24) with the 6-th order quadrature proposed by Kapur and Rokhlin [63]. The resulting linear system is notoriously difficult to solve iteratively.

Again, we constructed a HODLR approximation using the proxy surface technique (see, e.g., [9, Chapter 17]) on a CPU and copied the HODLR matrix to the GPU. We applied our HODLR solver and the block-sparse solver, and the results are shown in Table V. The factorization time and the solution time are plotted in Fig. 8, and Fig. 9 shows the floating point operations per second (Flop/s). Due to the oscillatory nature of the fundamental solution of the Helmholtz operator, the numerical ranks from low-rank compressions are higher than those in section IV-B associated with the Laplace operator for the same accuracy. As a result, the costs for solving (24) are generally higher. However, the observations are similar to before: (1) the costs of our GPU solver scaled nearly linearly; (2) our GPU solver achieved significant speedups over the parallel block-sparse solver (the factorization of the latter is faster than that of the sequential method); (3) we obtained a fast direct solver with highly accurate low-rank compressions as shown in Table Va. When relatively low-accuracy compressions were employed, we computed a reasonably accurate preconditioner much faster using much less memory.

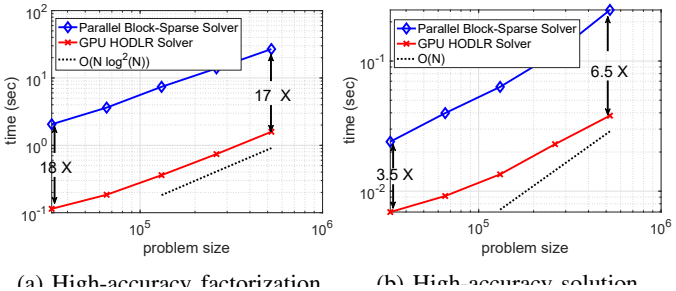
TABLE V: *Results for solving* (24) *with* $\eta = \kappa = 100$*, discretized with a 6-th order quadrature.*

N	Serial HODLR Solver			Serial Block-Sparse Solver			Parallel Block-Sparse Solver			GPU HODLR Solver			relres
	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	
$2^{15} = 32,768$	4.53e+0	1.92e-1	0.81	4.08e+0	8.28e-2	0.28	2.05e+0	2.40e-2	1.21	1.14e-1	6.91e-3	0.81	2.02e-9
$2^{16} = 65,536$	1.18e+1	4.86e-1	1.70	6.40e+0	1.51e-1	0.51	3.63e+0	3.98e-2	2.08	1.85e-1	9.18e-3	1.70	1.34e-9
$2^{17} = 131,072$	2.66e+1	1.01e-0	3.58	1.10e+1	2.87e-1	0.96	7.39e+0	6.33e-2	3.97	3.61e-1	1.35e-2	3.58	1.67e-9
$2^{18} = 262,144$	6.31e+1	2.15e-0	7.48	1.99e+1	5.69e-1	1.84	1.39e+1	1.14e-1	7.55	7.42e-1	2.29e-2	7.48	7.23e-10
$2^{19} = 524,288$	1.45e+2	5.22e-0	15.7	3.75e+1	1.12e-0	3.59	2.68e+1	2.47e-1	14.7	1.59e-0	3.80e-2	15.7	1.02e-9

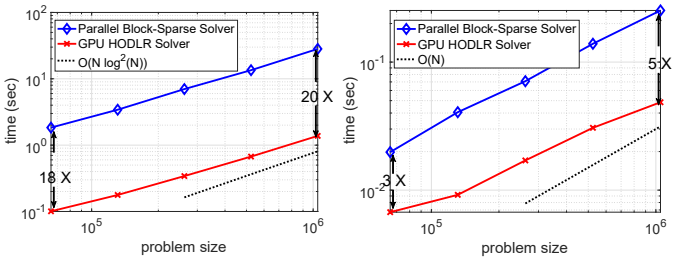
(a) *High-accuracy fast direct solver*

N	Serial HODLR Solver			Serial Block-Sparse Solver			Parallel Block-Sparse Solver			GPU HODLR Solver			relres
	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	t_f	t_s	mem	
$2^{15} = 32,768$	2.94e+0	1.57e-1	0.58	1.74e+0	4.31e-2	0.14	1.12e+0	1.73e-2	0.76	6.24e-2	4.44e-3	0.58	1.25e-4
$2^{16} = 65,536$	6.63e+0	3.24e-1	1.17	2.61e+0	8.72e-2	0.25	1.84e+0	1.98e-2	1.24	1.00e-1	6.73e-3	1.17	1.98e-4
$2^{17} = 131,072$	1.51e+1	6.71e-1	2.37	4.10e+0	1.37e-1	0.45	3.42e+0	4.06e-2	2.37	1.77e-1	9.19e-3	2.37	3.04e-4
$2^{18} = 262,144$	3.45e+1	1.51e-0	4.83	7.42e+0	2.69e-1	0.86	6.99e+0	7.09e-2	4.44	3.42e-1	1.71e-2	4.83	3.62e-4
$2^{19} = 524,288$	7.76e+1	3.19e-0	9.83	1.41e+1	5.24e-1	1.68	1.35e+1	1.39e-1	8.94	6.72e-1	3.07e-2	9.83	3.99e-4
$2^{20} = 1,048,576$	1.75e+2	6.92e-0	19.8	2.72e+1	1.05e-0	3.32	2.82e+1	2.53e-1	17.5	1.38e-0	4.86e-2	19.8	7.21e-4

(b) *Low-accuracy robust preconditioner*

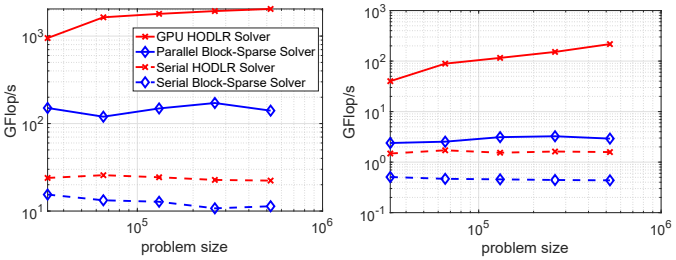


(a) High-accuracy factorization (b) High-accuracy solution



(c) Low-accuracy factorization (d) Low-accuracy solution

Fig. 8: Speedups for solving (24) discretized with a 6-th order quadrature.



(a) High-accuracy factorization (b) High-accuracy solution

Fig. 9: GFlop/s for solving (24). For the block-sparse solver, only the numerical factorization phase is considered in Fig. 9a.

V. Generalizations and Conclusions

We introduce algorithms for factorizing a HODLR matrix and for solving linear systems using the factorization on a GPU. The algorithms are based on a new data structure of storing the HODLR matrix and leverage efficient batched dense linear algebra kernels. We benchmarked the performance of our codes on kernel matrices and discretized BIEs, for which we constructed fast solvers with different levels of accuracies. In our numerical experiments, our GPU solver achieved significant speedups against reference methods.

Appendix

Off-diagonal ranks in HODLR approximations

Below are the ranks of off-diagonal blocks from level 1 to the leaf level:

- Table III, $N = 2^{21}$ (15 tree levels):
  56 54 45 52 44 30 41 38 38 25 33 24 22 19 18.
- Table IVa, $N = 2^{22}$ (16 tree levels):
  24 22 15 14 13 13 13 13 14 14 15 16 16 17 17 18.
- Table IVb, $N = 2^{24}$ (18 tree levels):
  1 1 1 2 3 3 4 4 5 5 6 7 7 8 8 9 10 11.
- Table Va, $N = 2^{19}$ (13 tree levels):
  225 134 97 69 54 46 41 39 37 35 33 31 29.
- Table Vb, $N = 2^{20}$ (14 tree levels):
  166 92 63 39 28 22 19 17 17 17 17 17 17 17.

References

[1] S. Ambikasaran and E. Darve, "An O (N log N) fast direct solver for partial hierarchically semi-separable matrices," *Journal of Scientific Computing*, vol. 57, no. 3, pp. 477–501, 2013.

[2] A. Aminfar, S. Ambikasaran, and E. Darve, "A fast block low-rank dense solver with applications to finite-element matrices," *Journal of Computational Physics*, vol. 304, pp. 170–188, 2016.

[3] A. G. Gray and A. W. Moore, "N-Body problems in statistical learning," *Advances in neural information processing systems*, pp. 521–527, 2001.

[4] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, vol. 36, no. 3, pp. 1171–1220, 2008.

[5] S. Ambikasaran, *Fast algorithms for dense numerical linear algebra and applications*. Stanford University, 2013.

[6] J. Y. Li, S. Ambikasaran, E. F. Darve, and P. K. Kitanidis, "A kalman filter powered by-matrices for quasi-continuous data assimilation problems," *Water Resources Research*, vol. 50, no. 5, pp. 3734–3749, 2014.

[7] S. Ambikasaran, M. O'Neil, and K. R. Singh, "Fast symmetric factorization of hierarchical matrices with applications," *arXiv preprint arXiv:1405.0223*, 2014.

[8] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, "Fast direct methods for Gaussian processes," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 252–265, 2015.

[9] P.-G. Martinsson, *Fast direct solvers for elliptic PDEs*. SIAM, 2019.

[10] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[11] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "Superfast multifrontal method for large structured linear systems of equations," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 1382–1411, 2010.

[12] P.-G. Martinsson, "A fast direct solver for a class of elliptic partial differential equations," *J. Sci. Comput.*, vol. 38, no. 3, p. 316–330, mar 2009. [Online]. Available: https://doi.org/10.1007/s10915-008-9240-6

[13] H. Ltaief, J. Cranney, D. Gratadour, Y. Hong, L. Gatineau, and D. Keyes, "Meeting the real-time challenges of ground-based telescopes using low-rank matrix computations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.

[14] P. R. Amestoy, A. Buttari, J.-Y. L'excellent, and T. Mary, "Performance and scalability of the block low-rank multifrontal factorization on multicore architectures," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 1, pp. 1–26, 2019.

[15] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, "Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures," in *International Supercomputing Conference*. Springer, 2017, pp. 22–40.

[16] N. Al-Harthi, R. Alomairy, K. Akbudak, R. Chen, H. Ltaief, H. Bagci, and D. Keyes, "Solving acoustic boundary integral equations using high performance tile low-rank lu factorization," in *International Conference on High Performance Computing*. Springer, 2020, pp. 209–229.

[17] W. Boukaram, S. Zampini, G. Turkiyyah, and D. Keyes, "H2opus-tlr: High performance tile low rank symmetric factorizations using adaptive randomized approximation," *arXiv preprint arXiv:2108.11932*, 2021.

[18] W. Hackbusch, "A sparse matrix arithmetic based on H-matrices. part I: Introduction to H-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[19] W. Hackbusch and B. N. Khoromskij, "A sparse H-matrix arithmetic. part ii: Application to multi-dimensional problems," *Computing*, vol. 64, no. 1, p. 21–47, Jan. 2000.

[20] W. Hackbusch and S. Börm, "Data-sparse approximation by adaptive H^2-matrices," *Computing*, vol. 69, no. 1, pp. 1–35, 2002.

[21] P.-G. Martinsson and V. Rokhlin, "A fast direct solver for boundary integral equations in two dimensions," *Journal of Computational Physics*, vol. 205, no. 1, pp. 1–23, 2005.

[22] S. Chandrasekaran, M. Gu, and T. Pals, "A fast ULV decomposition solver for hierarchically semiseparable representations," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[23] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "Fast algorithms for hierarchically semiseparable matrices," *Numerical Linear Algebra with Applications*, vol. 17, no. 6, pp. 953–976, 2010.

[24] L. Grasedyck, R. Kriemann, and S. Le Borne, "Domain decomposition based H-LU preconditioning," *Numerische Mathematik*, vol. 112, no. 4, pp. 565–600, 2009.

[25] R. Kriemann, "H-LU factorization on many-core systems," *Computing and Visualization in Science*, vol. 16, no. 3, pp. 105–117, 2013.

[26] K. L. Ho and L. Ying, "Hierarchical interpolative factorization for elliptic operators: integral equations," *Comm. Pure Appl. Math*, vol. 69, no. 7, pp. 1314–1353, 2016.

[27] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 1–35, 2016.

[28] P. Coulier, H. Pouransari, and E. Darve, "The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 3, pp. A761–A796, 2017.

[29] V. Minden, K. L. Ho, A. Damle, and L. Ying, "A recursive skeletonization factorization based on strong admissibility," *Multiscale Modeling & Simulation*, vol. 15, no. 2, pp. 768–796, 2017.

[30] J. Xia, "Multi-layer hierarchical structures," *CSIAM Transaction of Applied Mathematics*, vol. 2, pp. 263–296, 2021.

[31] Y. Liu, P. Ghysels, L. Claus, and X. S. Li, "Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S367–S391, 2021.

[32] D. Sushnikova, L. Greengard, M. O'Neil, and M. Rachh, "FMM-LU: A fast direct solver for multiscale boundary integral equations in three dimensions," *arXiv preprint arXiv:2201.07325*, 2022.

[33] S. Ambikasaran, K. R. Singh, and S. S. Sankaran, "HODLRlib: A library for hierarchical matrices," *Journal of Open Source Software*, vol. 4, no. 34, p. 1167, 2019.

[34] S. Massei, L. Robol, and D. Kressner, "hm-toolbox: Matlab software for HODLR and HSS matrices," *SIAM Journal on Scientific Computing*, vol. 42, no. 2, pp. C43–C68, 2020.

[35] Y. Dong and P.-G. Martinsson, "Simpler is better: A comparative study of randomized algorithms for computing the cur decomposition," *arXiv preprint arXiv:2104.05877*, 2021.

[36] L. Lin, J. Lu, and L. Ying, "Fast construction of hierarchical matrix representation from matrix–vector multiplication," *Journal of Computational Physics*, vol. 230, no. 10, pp. 4071–4087, 2011.

[37] P.-G. Martinsson, "Compressing rank-structured matrices via randomized sampling," *SIAM Journal on Scientific Computing*, vol. 38, no. 4, pp. A1959–A1986, 2016.

[38] J. Levitt and P.-G. Martinsson, "Linear-complexity black-box randomized compression of hierarchically block separable matrices," *arXiv preprint arXiv:2205.02990*, 2022.

[39] I. D. Fernando, S. Jayasena, M. Fernando, and H. Sundar, "A scalable hierarchical semi-separable library for heterogeneous clusters," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 513–522.

[40] P.-G. Martinsson, "A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1251–1274, 2011.

[41] W. Boukaram, G. Turkiyyah, and D. Keyes, "Randomized GPU algorithms for the construction of hierarchical matrices from matrix-vector operations," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. C339–C366, 2019.

[42] 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. 1–14.

[43] D. Y. Chenhan, S. Reiz, and G. Biros, "Distributed-memory hierarchical compression of dense SPD matrices," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 183–197.

[44] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[45] K. L. Ho and L. Greengard, "A fast direct solver for structured linear systems by recursive skeletonization," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. A2507–A2532, 2012.

[46] S. Ambikasaran and E. Darve, "The inverse fast multipole method," *arXiv preprint arXiv:1407.1572*, 2014.

[47] T. Takahashi, C. Chen, and E. Darve, "Parallelization of the inverse fast multipole method with an application to boundary element method," *Computer Physics Communications*, vol. 247, p. 106975, 2020.

[48] H. Pouransari, P. Coulier, and E. Darve, "Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation," *SIAM Journal on Scientific Computing*, vol. 39, no. 3, pp. A797–A830, 2017.

[49] D. A. Sushnikova and I. V. Oseledets, ""compress and eliminate" solver for symmetric positive definite sparse matrices," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. A1742–A1762, 2018.

[50] C. Chen, H. Pouransari, S. Rajamanickam, E. G. Boman, and E. Darve, "A distributed-memory hierarchical solver for general sparse linear systems," *Parallel Computing*, vol. 74, pp. 49–64, 2018.

[51] C. Chen, L. Cambier, E. G. Boman, S. Rajamanickam, R. S. Tuminaro, and E. Darve, "A robust hierarchical solver for ill-conditioned systems

with applications to ice sheet modeling," *Journal of Computational Physics*, vol. 396, pp. 819–836, 2019.
[52] J. Rotne and S. Prager, "Variational treatment of hydrodynamic interaction in polymers," *The Journal of Chemical Physics*, vol. 50, no. 11, pp. 4831–4837, 1969.
[53] H. Yamakawa, "Transport properties of polymer chains in dilute solution: hydrodynamic interaction," *The Journal of Chemical Physics*, vol. 53, no. 1, pp. 436–443, 1970.
[54] R. Kress, V. Maz'ya, and V. Kozlov, *Linear integral equations*. Springer, 1989, vol. 82.
[55] W. McLean and W. C. H. McLean, *Strongly elliptic systems and boundary integral equations*. Cambridge university press, 2000.
[56] A. Gillman, P. M. Young, and P.-G. Martinsson, "A direct solver with O(N) complexity for integral equations on one-dimensional domains," *Frontiers of Mathematics in China*, vol. 7, no. 2, pp. 217–247, 2012.
[57] T. A. Davis and I. S. Duff, "An unsymmetric-pattern multifrontal method for sparse lu factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 1, pp. 140–158, 1997.
[58] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.
[59] O. Schenk, K. Gärtner, and W. Fichtner, "Efficient sparse lu factorization with left-right looking strategy on shared memory multiprocessors," *BIT Numerical Mathematics*, vol. 40, no. 1, pp. 158–176, 2000.
[60] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.
[61] J. Kwack, G. Bauer, and S. Koric, "Performance test of parallel linear equation solvers on blue waters–cray xe6/xk7 system," in *Preceedings of the Cray Users Group Meeting (CUG2016), London, England*, 2016.
[62] K. Świrydowicz, E. Darve, W. Jones, J. Maack, S. Regev, M. A. Saunders, S. J. Thomas, and S. Peleš, "Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers," *Parallel Computing*, p. 102870, 2021.
[63] S. Kapur and V. Rokhlin, "High-order corrected trapezoidal quadrature rules for singular functions," *SIAM Journal on Numerical Analysis*, vol. 34, no. 4, pp. 1331–1356, 1997.