Adaptive Sketching Based Construction of H2 Matrices on GPUs

Wajih Halim Boukaram*, Yang Liu†, Pieter Ghysels‡, Xiaoye Sherry Li§
Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA, USA
*wajih.boukaram@lbl.gov, †liuyangzhuan@lbl.gov, ‡pghysels@lbl.gov, §xsli@lbl.gov

***Abstract*—We develop a novel linear-complexity bottom-up sketching-based algorithm for constructing a $\mathcal{H}^2$ matrix, and present its high performance GPU implementation. The construction algorithm requires both a black-box sketching operator and an entry evaluation function. The novelty of our GPU approach centers around the design and implementation of the above two operations in batched mode on GPU with accommodation for variable-size data structures in a batch. The batch algorithms minimize the number of kernel launches and maximize the GPU throughput. When applied to covariance matrices, volume IE matrices and $\mathcal{H}^2$ update operations, our proposed GPU implementation achieves up to $13\times$ speedup over our CPU implementation, and up to $1000\times$ speedup over an existing GPU implementation of the top-down sketching-based algorithm from the H2Opus library. It also achieves a $660\times$ speedup over an existing sketching-based $\mathcal{H}$ construction algorithm from the ButterflyPACK library. Our work represents the first GPU implementation of the class of bottom-up sketching-based $\mathcal{H}^2$ construction algorithms.**

***Index Terms*—$\mathcal{H}^2$-matrix, randomization, adaptive sketching, GPU**

I. Introduction

Many large-scale dense matrices from scientific and engineering applications exhibit low-rank structure after proper hierarchical matrix partitioning. Such low-rank structure can be exploited by hierarchical matrix techniques to enable fast matrix-vector multiplication and matrix inversion in quasi-linear time. Examples include integral equation methods for acoustics, electromagnetics [1]–[3], Stokes flows [4] and charged particle systems [5], differential equation-based PDE solvers [6], [7], machine learning methods like kernel ridge regression [8] and Gaussian processes [9], and various other structured matrices, e.g., Toeplitz and Cauchy [10].

There exists a broad family of hierarchical matrix techniques, including the $\mathcal{H}/\mathcal{H}^2$ formats [1]–[3], the hierarchically off-diagonal low-rank format (HODLR) [11], the hierarchically semi-separable format (HSS) [12] or hierarchically block separable format (HBS) [13], the inverse fast multipole method (IFMM) [14] and the hierarchical interpolative factorization (HIF) algorithms [15]. These formats can be characterized by the so-called admissibility condition which determines how much separated interaction can be low-rank compressed. The optimal choice of hierarchical format depends on the particular application, including the dimensionality and discretization scheme. For high-dimensional problems, weak-admissibility-based formats such as HODLR, HSS, HBS typically cannot achieve quasi-linear complexities with the exception of HIF, which however only has been demonstrated with regular-grid-based discretization. On the other hand, strong-admissibility-based formats can attain quasi-linear (e.g. $\mathcal{H}$) and linear complexities (e.g. $\mathcal{H}^2$ and IFMM) for high-dimensional problems. That being said, they usually show larger prefactors and/or pose challenges for scalable parallel implementations.

This paper focuses on efficient algorithms for the construction of the $\mathcal{H}^2$ format. Just like the other hierarchical matrix formats, a $\mathcal{H}^2$ matrix can be efficiently constructed by assuming that (a) any matrix entry can be computed quickly on-the-fly [3], [16]–[18] or (b) a fast black-box sketching operator is available. Here (a) is commonly encountered for compressing forward operators in integral equations and kernel matrices. Existing codes include HLIBpro [3], [18], H2Pack [16], ASKIT [19], GOFMM [20], and GPU implementations like H2Opus [17] and hmglib [21]. They typically leverage adaptive cross approximation, proxy surface, or preselected skeletons to construct the $\mathcal{H}^2$ matrix. On the other hand, (b) is often encountered for compressing frontal matrices in sparse multifrontal solvers, trace estimation in Bayesian optimization, or low-rank updating an existing $\mathcal{H}^2$ matrix. Unlike (a), there exist fewer known sketching-based algorithms and implementations for (b), which include the top-down peeling algorithms [17], [22], [23] and the more recent bottom-up algorithm [24].

We focus on algorithms based on assumption (b) in this paper. We propose a partially black-box sketching-based $\mathcal{H}^2$ construction algorithm requiring fewer samples compared to existing algorithms and describe an efficient GPU implementation, particularly useful for accelerating $\mathcal{H}^2$ arithmetic in sparse multifrontal solvers or Schur complement-based updates. It is worth mentioning that once the $\mathcal{H}^2$ matrix has been constructed, efficient (i.e., low-prefactor) inversion algorithms have also been recently developed [25], [26] and parallelized [27], [28]. But there is no GPU algorithm for inversion. This current paper describes the construction phase for $\mathcal{H}^2$ on GPU. In a future paper, we will describe the GPU algorithm for $\mathcal{H}^2$ inversion.

Our main contributions can be summarized as follows:

- We develop a novel partially black-box $\mathcal{H}^2$ matrix construction algorithm with linear complexity, which extends the bottom-up algorithm in [7], [29] from weakly-admissible HSS to strongly-admissible $\mathcal{H}^2$ and permits adaptive sketching. Compared with the top-down algorithms in [17], [23], the proposed algorithm requires much fewer samples and is asymptotically faster.

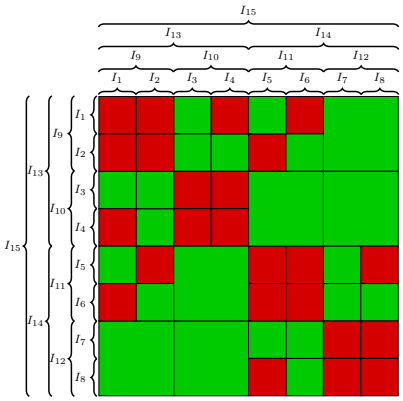


Fig. 1: The leaves of the hierarchical matrix tree forming a block partitioning of the matrix. Red blocks represent inadmissible leaves and green blocks represent admissible blocks. Row and columns indices are hierarchically clustered into a cluster tree I such that pairs of clusters define blocks within the matrix.

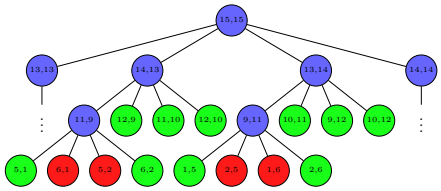


Fig. 2: The matrix tree for the hierarchical matrix in Fig. 1 representing the inadmissible blocks in blue, the admissible leaves in green and the inadmissible leaves in red. In general, the matrix tree is not a complete tree. The ellipsis represent a complete subtree of the tree and are omitted for brevity.

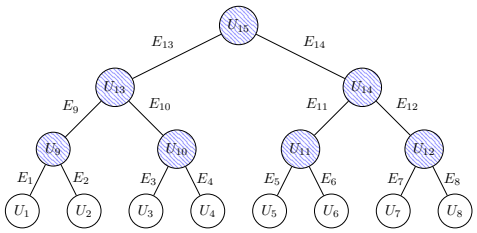


Fig. 3: The basis tree for the hierarchical matrix in Fig. 1 where leaves U_τ are stored explicitly and the shaded inner nodes are implicitly represented by the nested basis property using the transfer matrices E.

- We develop a GPU implementation of the construction algorithm relying on batched dense linear algebra kernels and batched entry extraction routines. This represents the first parallel GPU implementation of the partially black-box or fully black-box [24] bottom-up construction algorithms.
- We demonstrate the computational efficiency of the proposed algorithm by compressing integral equations and covariance matrices, and recompressing $\mathcal{H}^2$ matrices with low-rank updates. Our GPU implementation achieves up to $13\times$ speedup over our CPU implementation, and up to $1000\times$ speedup over the GPU implementation of a top-down sketching-based algorithm in H2Opus. It's also worth mentioning that our CPU and GPU implementations share the same code base due to the use of Thrust.

The remainder of the paper is structured as follows: Section II introduces $\mathcal{H}^2$ matrices and associated preliminaries including cluster tree and interpolative decomposition. Sections III and IV describe the main contributions, the adaptive sketching construction and the high-performance GPU implementation, respectively. Performance results are reported in Section V and we conclude in Section VI.

II. Preliminaries

A. Hierarchical Matrices

Hierarchical matrices aim to provide an efficient representation of dense matrices that are data sparse, where certain blocks within these dense matrices can be well approximated by a low-rank form. Many different variants of hierarchical matrices have been developed over the years, primarily differentiating themselves on the block partitioning used and the representation of the basis vectors used to approximate the blocks. The block partitioning is typically determined by first hierarchically clustering the indices of the matrix K into a cluster tree I, and then performing a dual tree traversal on I. The traversal generates pairs of clusters (s,t) that are tested against an admissibility condition that determines whether the matrix block defined by the cluster pair can be approximated well by a low-rank matrix. We consider the so-called general admissibility condition `adm` which determines the compressibility of a block based on the distance `Dist` between the bounding boxes of the cluster pair (s,t) and the average of their diameters D:

$$\texttt{adm}(s,t) = 1, \; if \; \frac{D(s)+D(t)}{2} \leq \eta \texttt{Dist}(s,t) \qquad (1)$$

Typically $\eta \geq 1$ indicates the so-called weak admissibility and $\eta \leq 0.5$ indicates the so-called strong admissibility. The general admissibility condition is used to perform a dual tree traversal. The traversal produces a matrix tree where each node is a cluster pair (s,t) at the same level. If a cluster pair is deemed inadmissible, the dual tree traversal continues on their four children until the block defined by the pair is sufficiently small and thus stored in its original dense form. The full set of leaves of this matrix tree then define the block partitioning of the matrix. Fig. 1 illustrates the cluster tree I whose dual traversal with a general admissibility condition produced a block partitioning of the matrix where admissible leaves are shown in green and inadmissible leaves are shown in red. The corresponding matrix tree is shown in Fig. 2 with the complete subtrees $K_{13,13}$ and $K_{14,14}$ on the diagonal omitted for brevity. All the leaves within a level of the matrix tree can be viewed as a block sparse matrix, and one important property of hierarchical matrices is that the number of blocks in a row of each level's block sparse matrix is bounded by

a constant that does not grow with the problem size. This constant is called the sparsity constant C_{sp}.

More specifically, Fig. 4(a)-(b) shows the block partitioning of a matrix associated with a set of $N = 2^{15}$ 3D geometry points, using the admissibility parameter $\eta = 0.5, 0.7$. Note that smaller η leads to more refined partitioning of the off-diagonal blocks, and hence larger sparsity constants C_{sp}.

Let us denote the block of a matrix K determined by a cluster pair (s,t) as $K_{s,t}$, and the set of clusters that form inadmissible pairs with a cluster τ as $\mathcal{N}_\tau$. The set of clusters (1) that form admissible pairs with cluster τ and (a) whose parents form inadmissible pairs with parent of τ, are denoted as $\mathcal{F}_\tau$. These notations are summarized in Table I. We also use MATLAB notation when convenient. Now that we've determined the desired block partitioning, we would like to represent the admissible blocks $K_{s,t}$ in a low-rank form. The $\mathcal{H}$-matrix variant represents each $m \times n$ block $K_{s,t}$ of rank k as the outer product $K_{s,t} = U_{s,t}V_{s,t}^T$, where $U_{s,t}$ and $V_{s,t}$ are $m \times k$ and $n \times k$ matrices respectively, leading to $O(n \log n)$ storage complexity for the matrix. On the other hand, the $\mathcal{H}^2$-matrix variant use a nested basis to achieve $O(n)$ storage complexity. Instead of storing independent U and V matrices for each block, $\mathcal{H}^2$-matrices use a common basis for the block rows/columns defined by each cluster in the cluster tree, introducing a smaller $k \times k$ coupling matrix B for each block and representing each block as $K_{s,t} = U_s B_{s,t} V_t^T$. For simplicity, we assume K is symmetric and real-valued in the rest of this paper, leading to $V_t = U_s^T$. However, our algorithm can be easily extended to un-symmetric or complex-valued matrices.

The basis for leaf nodes in the cluster tree are stored explicitly, and the basis for an inner node τ of the cluster tree is defined in terms of the basis of its children τ_1 and τ_2 using transfer matrices E, resulting in a nested basis:

$$U_\tau = \begin{bmatrix} U_{\tau_1} & \\ & U_{\tau_2} \end{bmatrix} \begin{bmatrix} E_{\tau_1} \\ E_{\tau_2} \end{bmatrix} \tag{2}$$

Fig. 3 shows a basis tree for the clusters in Fig. 1, where the clear nodes at the leaf level are stored explicitly and the shaded internal nodes are represented implicitly using the nested basis property.

B. Interpolative Decomposition

The interpolative decomposition (ID) aims to compute a factorization of an $m \times n$ matrix A such that A can be approximated as a linear combination of a set S of selected columns: $A \approx A(:,S)X$, where the rank $k = \text{card}(S)$ is usually selected to satisfy some approximation threshold ϵ. We refer to this ID as the column ID. The column ID can be computed using the column pivoted QR decomposition, where a column permutation P of A is factored into an orthogonal factor Q and a triangular factor R: $AP = QR$. The column ID can then be computed as follows:

$$\begin{aligned} AP = QR &= \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 & R_2 \\ 0 & R_3 \end{bmatrix} \\ &= Q_1 \begin{bmatrix} R_1 & R_2 \end{bmatrix} + Q_2 \begin{bmatrix} 0 & R_3 \end{bmatrix} \\ &\approx Q_1 R_1 \begin{bmatrix} I & R_1^{-1}R_2 \end{bmatrix} = A(:,S) \begin{bmatrix} I & T \end{bmatrix} \end{aligned} \tag{3}$$

By discarding the lower right triangular factor R_3 when its norm becomes small enough to guarantee the approximation threshold ϵ for A, we can obtain the interpolation matrix $T = R_1^{-1}R_2$. Similarly, the row ID for A is defined as $PA = \begin{bmatrix} I & T \end{bmatrix}^T A(S,:)$, which is typically computed via the column ID of A^T. When referring to matrix column or row indices, we will refer to the set of selected indices S as the skeletonization indices and the remaining unselected indices as the redundant indices R[1].

TABLE I: Notation

τ	A node in the cluster tree
U_τ	Basis matrix for a leaf node
E_τ	Transfer matrix for an inner node
$B_{s,t}$	Coupling matrix for admissible node pair (s,t)
$D_{s,t}$	Dense leaf matrix for inadmissible node pair (s,t)
$\mathcal{N}_\tau$	Set of clusters that form inadmissible pairs with τ
$\mathcal{F}_\tau$	Set of clusters that form admissible pairs with τ

III. Sketching Construction

In this section, we discuss the details of the construction of hierarchical matrices using the sketching algorithm, starting with the fixed sample version in Section III-A that assumes that the number of samples needed for construction is known beforehand. The proposed algorithm represents the extension of a sketching-based construction algorithm for the HSS matrix [29] to strongly-admissible $\mathcal{H}^2$ matrices. This algorithm is then generalized to the adaptive version in Section III-B that adds additional samples as needed to satisfy a compression threshold ϵ. We will assume that the matrix is symmetric to simplify the discussion, as non-symmetric matrices are a straightforward modification to the algorithm. We also assume that a hierarchical block partitioning of the matrix that would allow for low-rank compression is already computed.

A. Construction with Fixed Sample Size

Let us first assume that we know the representative rank r of the hierarchical matrix that we're constructing and specify an oversampling parameter p designed to assure with high probability that the total number of samples $d = r + p$ is sufficient to guarantee the construction to the required accuracy. To simplify the discussion, let us also assume that the indices of the matrix are already sorted so that all redundant indices R within a cluster come before all the skeletonization indices S. In other words, we drop the permutation matrix P in column and row IDs in what follows. The randomized construction algorithm (See Algorithm 1) requires two inputs:

[1] R is used from this point onward to denote the redundant indices.

Algorithm 1 Proposed $\mathcal{H}^2$ construction algorithm for a matrix K based on sketching (permutation matrices are not shown)

Input: Sample block size d, a hierarchical partitioning of the blocks of the matrix of L levels, a relative compression tolerance ϵ, a black-box function $Y = K_{blk}(\Omega)$ that can compute $Y = K\Omega$ with a random matrix $\Omega \in \mathbb{R}^{N\times d}$ in $O(Nd)$ time, and a function to evaluate any subblock $K_{s,t}$.
Output: Skeletonization indices $\tilde{I}_\tau$ for each node τ. $\mathcal{H}^2$ matrix $K_\mathcal{H}$ with, for each node τ, U_τ, $D_{\tau,b}$ ($b \in \mathcal{N}_\tau$), $B_{\tau,b}$ ($b \in \mathcal{F}_\tau$) at the leaf level, and E_{τ_1},E_{τ_2}, $B_{\tau,b}$ ($b \in \mathcal{F}_\tau$) at higher levels.

1: $Y = K_{blk}(\Omega)$ with a random $\Omega \in \mathbb{R}^{N\times d}$ ▷ batchedRand
2: **for** level $l = 1, \ldots, L$ **do**
3: **if** $l = 1$ **then** ▷ Leaf node
4: **for** node τ at level l **do**
5: $\Omega^1_\tau = \Omega(I_\tau, :), Y^1_\tau = Y(I_\tau, :)$
6: **end for**
7: **for** node τ at level l **do**
8: $D_{\tau,b} = K(I_\tau, I_b)\ \forall b \in \mathcal{N}_\tau,$ ▷ batchedGen
9: $Y^{\text{loc}}_\tau = Y^1_\tau - \sum_{b\in\mathcal{N}_\tau} D_{\tau,b}\Omega^1_b$ ▷ batchedBSRGemm
10: **end for**
11: **while** $\exists\ \tau$ non-converged (via QR of Y^{loc}_τ) **do**
12: $\bar{Y} = K_{blk}(\bar{\Omega})$ with a random $\bar{\Omega} \in \mathbb{R}^{N\times d}$ ▷ batchedRand
13: $Y^{\text{loc}}_\tau, \Omega^l_\tau, \forall\tau$ at l = updateSamples($\bar{Y}, \bar{\Omega}, l$)
14: **end while**
15: **for** node τ at level l **do**
16: $Y^{\text{loc}}_\tau = U_\tau Y^{\text{loc}}_\tau(J_\tau, :)$ ▷ ID with ϵ_l. batchedID
17: $Y^{l+1}_\tau = Y^{\text{loc}}_\tau(J_\tau, :)$ ▷ batchedShrink
18: $\Omega^{l+1}_\tau = U^T_\tau \Omega^l_\tau$ ▷ batchedGemm
19: $\tilde{I}_\tau = I_\tau(J_\tau)$ ▷ Pick the J_τ indices from I_τ
20: **end for**
21: **else** ▷ Inner node
22: **for** node τ at level l **do**
23: Let ν_1 and ν_2 be the children of τ
24: $\bar{I}_\tau = [\tilde{I}_{\nu_1}, \tilde{I}_{\nu_2}], \Omega^l_\tau = \begin{bmatrix}\Omega^l_{\nu_1}\\ \Omega^l_{\nu_2}\end{bmatrix}, Y^l_\tau = \begin{bmatrix}Y^l_{\nu_1}\\ Y^l_{\nu_2}\end{bmatrix}$
25: **end for**
26: **for** node τ at level l **do**
27: $Y^{\text{loc}}_\tau = Y^l_\tau - \begin{bmatrix}\sum_{b\in\mathcal{F}_{\nu_1}} B_{\nu_1,b}\Omega^l_b\\ \sum_{b\in\mathcal{F}_{\nu_2}} B_{\nu_2,b}\Omega^l_b\end{bmatrix}$ ▷ batchedBSRGemm
28: **end for**
29: **while** $\exists\ \tau$ non-converged (via QR of Y^{loc}_τ) **do**
30: $\bar{Y} = K_{blk}(\bar{\Omega})$ with a random $\bar{\Omega} \in \mathbb{R}^{N\times d}$ ▷ batchedRand
31: $Y^{\text{loc}}_\tau, \Omega^l_\tau, \forall\tau$ at l = updateSamples($\bar{Y}, \bar{\Omega}, l$)
32: **end while**
33: **for** node τ at level l **do**
34: $Y^{\text{loc}}_\tau = \begin{bmatrix}E_{\tau_1}\\ E_{\tau_2}\end{bmatrix} Y^{\text{loc}}_\tau(J_\tau, :)$ ▷ ID with ϵ_l. batchedID
35: $Y^{l+1}_\tau = Y^{\text{loc}}_\tau(J_\tau, :)$ ▷ batchedShrink
36: $\Omega^{l+1}_\tau = \begin{bmatrix}E^T_{\tau_1} & E^T_{\tau_2}\end{bmatrix} \Omega^l_\tau$ ▷ batchedGemm
37: $\tilde{I}_\tau = \bar{I}_\tau(J_\tau)$ ▷ Pick the J_τ indices from $\bar{I}_\tau$
38: **end for**
39: **end if**
40: **for** node τ at level l **do**
41: $B_{\tau,b} = K(\tilde{I}_\tau, \tilde{I}_b)\ \forall b \in \mathcal{F}_\tau$ ▷ batchedGen
42: **end for**
43: **end for**

(a) a black box function that computes $Y = K\Omega$ with a random set of vectors $\Omega \in \mathbb{R}^{N\times d}$, and (b) a function used to evaluate a small number of matrix entries. The algorithm uses Y to recursively sketch the admissible blocks at each level of the $\mathcal{H}^2$ matrix, after subtracting out contribution to Y of the inadmissible blocks at each level via direct matrix entry evaluations. Note that for now one can ignore the while loops in grey at Lines 11 and 29, which will be explained in Section III-B in the context of adaptive sampling.

1) Construction at the leaf level: At the leaf level, tthe influence of the inadmissible leaf blocks is first subtracted out from the samples so that we are left with just the samples of the admissible part of the matrix. For each node τ, the dense blocks $D_{\tau,b}$, $b \in \mathcal{N}_\tau$ are evaluated using the index sets of its defining clusters I_τ, I_b, multiplied by the submatrices of the input vector corresponding to the cluster t, $\Omega^1_t = \Omega(I_t, :)$ and then subtracted from the samples submatrix $Y^1_\tau = Y(I_\tau, :)$. This is reflected by Line 9 of Algorithm 1: $Y^{\text{loc}}_\tau = Y^1_\tau - \sum_{b\in\mathcal{N}_\tau} D_{\tau,b}\Omega^1_b$. Fig. 4(c) shows the admissible blocks which contribute to each Y^{loc}_τ.

Performing the row ID on the samples Y^{loc}_τ for each cluster τ, $Y^{\text{loc}}_\tau = \begin{bmatrix}T_\tau & I\end{bmatrix}^T Y^{\text{loc}}_\tau(J_\tau, :)$, gives us an interpolation matrix T_τ that can be used as the interpolation matrix for the admissible block row/column for the cluster. The basis for cluster τ can thus be computed using the interpolation matrix as $U_\tau = \begin{bmatrix}T_\tau & I\end{bmatrix}^T$. See Line 16 of Algorithm 1. Let us define two block unit-triangular matrices:

$$W_\tau = \begin{bmatrix} I & -T^T_\tau \\ 0 & I \end{bmatrix} \quad \text{and} \quad Z_\tau = W^T_\tau = \begin{bmatrix} I & 0 \\ -T_\tau & I \end{bmatrix} \tag{4}$$

After scaling the block row $K(I_s, :)$ by W_s and block column $K(:, I_t)$ by Z_t, each admissible matrix block $K(I_s, I_t)$ is modified as

$$W_s K(I_s, I_t) Z_t \approx \begin{bmatrix} 0 & 0 \\ 0 & K(\tilde{I}_s, \tilde{I}_t) \end{bmatrix}, \tag{5}$$

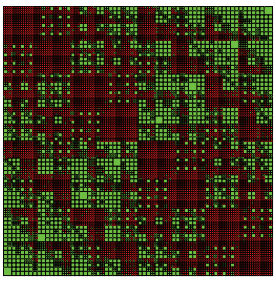
which can effectively remove the redundant portion of $K(I_s, I_t)$. The effect of this scaling can be seen for the first cluster in Fig. 4(d). As we skeletonize a cluster τ, we replace its original index set I_τ with those selected by the ID: $\tilde{I}_\tau$ (see Line 19 of Algorithm 1). This process can be performed in parallel for all clusters at the leaf level, leading to the leaf level being skeletonized as in 4(e).

Since W_τ and Z_τ are block unit-triangular matrices, inverting them involves simply flipping the sign of the interpolation matrix, giving us the approximation of the block

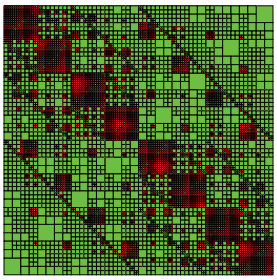
$$\begin{aligned} K(I_s, I_t) &\approx \begin{bmatrix} I & T^T_s \\ 0 & I \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & K(\tilde{I}_s, \tilde{I}_t) \end{bmatrix} \begin{bmatrix} I & 0 \\ T_t & I \end{bmatrix} \\ &= \begin{bmatrix} T^T_s \\ I \end{bmatrix} K(\tilde{I}_s, \tilde{I}_t) \begin{bmatrix} T_t & I \end{bmatrix} = U_s K(\tilde{I}_s, \tilde{I}_t) U^T_t \end{aligned} \tag{6}$$

For each cluster s and $t \in \mathcal{F}_s$, the coupling matrix is computed by directly evaluating the matrix entries at the skeletonization indices of those clusters: $B_{s,t} = K(\tilde{I}_s, \tilde{I}_t)$. Note that for $t \notin \mathcal{F}_s$, $B_{s,t}$ is not explicitly formed and will be sketched at higher levels.

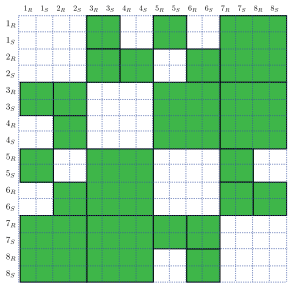
2) Construction at higher levels: To continue the skeletonization process beyond the leaf level, we must first ensure that we have samples of the remaining admissible skeletonized portion of the matrix. At the end of the skeletonization process



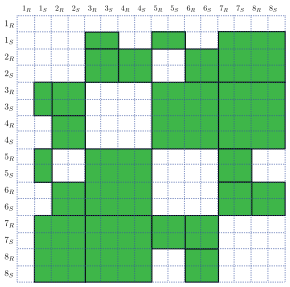
(a) $\eta = 0.5$



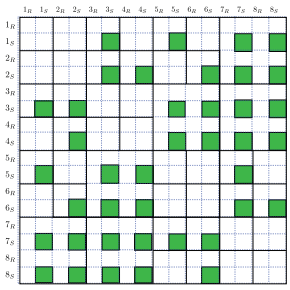
(b) $\eta = 0.7$



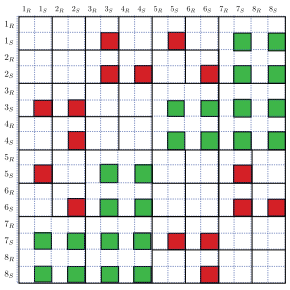
(c) Dense blocks removed



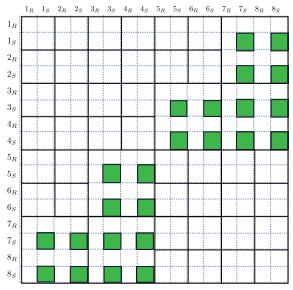
(d) Skeletonizing the first cluster



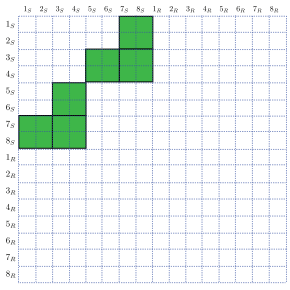
(e) All leaf level clusters skeletonized



(f) Leaf level coupling matrices computed and marked for removal in red



(g) Leaf level coupling matrices removed



(h) The remaining skeletonized blocks for the next level

Fig. 4: (a)-(b) Block partitioning of a hierarchical matrix for a 3D problem of size $N = 2^{15}$ with different η. (c)-(h) The skeletonization process for the leaf level of the $\mathcal{H}^2$-matrix in Fig. 1.

for the leaf level, the matrix K has been transformed by two block diagonal matrices W^1 and Z^1 where the diagonal blocks are the previously defined W_τ and Z_τ matrices, respectively. We define the transformed matrix K at level l to be $K^l = W^l K^{l-1} Z^l$ with $K^0 = K$. To continue the skeletonization at level l, we need to extract samples Y^l of K^l relying only on the data from the input and output vectors of the previous levels Y^{l-1}, Ω^{l-1} with $Y^0 = Y = K\Omega$ and $\Omega^0 = \Omega$. First, we transform the random input vectors using $(Z^l)^{-1}$ to obtain the next set of input vectors $\Omega^1 = (Z^l)^{-1}\Omega^{l-1}$. Using these input vectors on K^l gives us:

$$\begin{aligned} Y^l &= K^l\Omega^l = W^l K^{l-1} Z^l (Z^l)^{-1}\Omega^{l-1} \\ &= W^l K^{l-1}\Omega^{l-1} = W^l Y^{l-1} \end{aligned} \tag{7}$$

This allows us to represent the samples at level l, Y^l, as a transformation of the samples of the previous level Y^{l-1}. In fact, one can see from (5) that K^l has been significantly sparsified compared with K^{l-1} and one doesn't need to form Y^l and Ω^l in full. Instead, substituting (6) into (7) reveals that one only needs to compute subvectors of Y^l and Ω^l once blocks at level $l-1$ has been skeletonized.

We can see this for the leaf level and higher levels as the follows: (a) At the leaf level, we can compute them as $Y^2_\tau = Y^{\text{loc}}_\tau(J_\tau, :)$ (Line 17 of Algorithm 1) and $\Omega^2_\tau = U^T_\tau \Omega^1_\tau$ (Line 18 of Algorithm 1). (b) At a higher level l, the samples and random vectors for cluster τ are formed as $\Omega^l_\tau = \begin{bmatrix} \Omega^l_{\nu_1} \\ \Omega^l_{\nu_2} \end{bmatrix}$ and $Y^l_\tau = \begin{bmatrix} Y^l_{\nu_1} \\ Y^l_{\nu_2} \end{bmatrix}$, respectively. Here ν_1 and ν_2 are the children of ν. The contribution to Y^l_τ for $\mathcal{F}_{\nu_1}$ and $\mathcal{F}_{\nu_2}$ is subtracted out as $Y^{\text{loc}}_\tau = Y^l_\tau - \begin{bmatrix} \sum_{b\in\mathcal{F}_{\nu_1}} B_{\nu_1,b}\Omega^l_b \\ \sum_{b\in\mathcal{F}_{\nu_2}} B_{\nu_2,b}\Omega^l_b \end{bmatrix}$ (see Line 27 of Algorithm 1). Note that the coupling matrices $B_{\nu_1,b}$ and $B_{\nu_2,b}$ have been explicitly formed at the previous level. As an example, see Fig. 4(f) where $B_{\nu_1,b}$ and $B_{\nu_2,b}$ have been marked in red, and 4(g) for the blocks that contribute to Y^{loc}_τ. Then the transfer matrices in (2), E_{ν_1} and E_{ν_2}, as well as the skeleton indices $\tilde{I}_\tau$ can be computed from the row ID of Y^{loc}_τ at Line 34 of Algorithm 1. Note that a row permutation matrix has been ignored at Line 34. Now we can compute the subvectors of Y^{l+1} and Ω^{l+1} as $Y^{l+1}_\tau = Y^{\text{loc}}_\tau(J_\tau, :)$ and $\Omega^{l+1}_\tau = \begin{bmatrix} E^T_{\tau_1} & E^T_{\tau_2} \end{bmatrix} \Omega^l_\tau$, respectively. See 4(h) for blocks that correspond to Y^{l+1} and Ω^{l+1}. Finally, for each cluster $b \in \mathcal{F}_\tau$, the coupling matrix is computed by directly evaluating the matrix entries at the skeletonization indices of those clusters: $B_{\tau,b} = K(\tilde{I}_\tau, \tilde{I}_b)$.

Algorithm 1 represents the extension of the sketching-based construction algorithm for the HSS matrix [29] to strongly-admissible $\mathcal{H}^2$ matrices. Therefore, we claim that computational complexity and error behavior of Algorithm 1 can be analyzed by extending corresponding analyses in‘ [29]. We leave the detailed analyses as future work. It is worth noting that HSS typically reveals large, non-constant ranks causing superlinear computational complexity for higher-dimensional problems, but the $\mathcal{H}^2$ matrix allows for linear CPU and memory complexity with small ranks. One can readily see that Algorithm 1 is an $O(N)$ algorithm assuming the rank r is a small constant (more precisely, $r = O(\log 1/\epsilon)$ and we

assume the tolerance is fixed in this paper), as it only requires $O(r^2N)$ time to generate samples Y and direct evaluation of $O(rN)$ matrix entries.

B. Construction With Adaptive Sampling

The number of samples needed to satisfy a specific relative error threshold ϵ for the construction of a hierarchical matrix is typically not know beforehand. A few changes to the fixed rank algorithm are needed to support adaptive construction. First, before performing ID to determine the skeletonization indices, we ensure that the current set of samples contain enough data to approximate the node (see the convergence test at Lines 11 and 29 of Algorithm 1). This can be achieved by computing the QR decomposition of a node's set of sample vectors Y_τ^{loc} and examining the smallest absolute value of the diagonal of the triangular factor. If this value is less than an absolute error threshold ϵ_{abs}, then we consider the node converged. To support a relative threshold, an approximate norm of the matrix can be provided via sketching and the absolute threshold ϵ_{abs} would simply be the product of the relative threshold ϵ and the norm. If not converged, we add more samples by $\bar{Y} = K_{blk}(\bar{\Omega})$ with a new random matrix $\bar{\Omega} \in \mathbb{R}^{N \times d}$. These new samples are used to update $\bar{Y} = K_{blk}(\bar{\Omega})$ and Ω_τ^l represented by the updateSamples function at Lines 13 and 31. When all nodes within a level have converged, then we can stop adding samples and move on to ID.

IV. GPU Implementation

We describe the proposed GPU implementation of the adaptive construction method of Algorithm 1 in Section IV-A, followed by a performance analysis in Section IV-B.

A. Adaptive GPU Sketching

In this subsection, we explain the GPU implementation of the proposed algorithm in Algorithm 1 in detail. First, we note that launching a kernel on the GPU incurs an overhead that can dominate an application's runtime if the kernel has a small compute workload. Likewise, individual memory allocation for each small kernel can hinder GPU performance. While $\mathcal{H}^2$-matrices provide asymptotically optimal storage and algorithmic complexity, they consist of small dense blocks whose individual kernel launch and memory allocation become impractical. Therefore, a more nuanced approach is required to achieve high performance on GPUs.

In our proposed implementation, the nodes of the trees are stored contiguously level by level to expose the parallelism in each level of the tree. Then most operations are split into two phases: a marshaling phase where data from the flattened trees relevant to the operation is gathered and a batched execution phase where batch routines carry out all operations using a single kernel call. Unless otherwise stated, the batch count is set to the number of nodes of a given level. The comments in blue-green in Algorithm 1 indicate the operations (or loops) that are implemented with GPU kernels. The marshaling routines are executed on the device using the Thrust library [30] while the majority of the batched routines are provided by the KBLAS [31] and MAGMA [32] libraries. Note that most of the batched operations involve non-uniform-sized matrices as the cluster sizes and ranks are not constant. One advantage of this approach is that the same code, with trivial modification, can run on either CPUs or GPUs. This is due to that Thrust has multiple parallel CPU backends and the batched routines can be easily implemented on the CPU using parallel OpenMP loops around single threaded BLAS and LAPACK routines. We remark that, when one executes the proposed algorithm on the GPU, all the data, the black-box function $K_{blk}(\cdot)$ and the entry evaluation function fully reside on GPUs.

We first modify the inputs of Algorithm 1 such that instead of a function to evaluate any subblock $K_{s,t}$ on the GPU, we require a function to evaluate a batch of subblocks on the GPU. We call this function batched entry generator and it's invoked to evaluate all $D_{\tau,b}$ or $B_{\tau,b}$ at a given level l with a single kernel launch (see Lines 8 and 41 marked by *batchedGen*). Next, the random matrices Ω or $\bar{\Omega}$ are generated in a single kernel and supplied to $K_{blk}(\cdot)$ to produce Y or $\bar{Y}$ on the GPU (see Lines 1, 12, 30 marked by *batchedRand*). To avoid large amounts of small memory allocations, the total amount needed per level is first determined using a Thrust parallel prefix sum on the block dimensions and then allocated in a single allocation per operation.

When computing Y_τ^{loc} from Y_τ^l to ensure that the samples only include the influence of the admissible blocks as in Fig. 4(c) and 4(g), we use a non-uniform batched block sparse row (BSR) matrix multiplication routine. For example, the BSR matrix would include the red dense blocks in Fig. 4(f). Since no GPU implementation for non-uniform blocks in a BSR matrix product currently exists, we take advantage of the sparsity constant described in Section II to split the operation into at most C_{sp} kernels each performing a batched non-uniform matrix-matrix multiplication using MAGMA. Each kernel works on marshaled data in a way that allows us to update the output vectors in parallel without resorting to atomic operations; that is to say that only one block from each row will be involved in each kernel launch, and since we have at most C_{sp} such kernels, the kernel launch overhead should not impact performance.

After obtaining a set of samples of the admissible part of the matrix, the convergence test checks if all nodes have converged to satisfy the error threshold ϵ_l using the method described in Section III-B. If the current set of samples prove to be insufficient, additional samples and input vectors are generated. The updateSamples function at Lines 13 and 31 sweeps any new samples and input vectors up the tree until it reaches the current level.

Once we've acquired a sufficient number of samples Y_τ^{loc}, a batched row ID determines the skeletonization indices. As the row ID is implemented via the column ID on the matrix transpose. The list of samples are first accumulated using a batch transpose to allow for more efficient memory access patterns on the GPU for a column pivoted QR. See Lines 16

and 34 marked by *batchedID*.

Finally, the input vectors Ω_τ^l are upswept to the next level using a batched matrix multiplication (see Lines 18 and 36 marked by *batchedGemm*) and the samples Y_τ^{loc} are upswept by first swapping the columns of the previously transposed samples which are then transposed again to their row skeletonized form Y_τ^{l+1} (see Lines 17 and 35 marked by *batchedShrink*).

B. Performance Analysis

Here we provide a brief analysis of the parallelization performance of Algorithm 1 on the GPU. As described in Section IV, the operations implemented on GPU are batchedRand, batchedBSRGemm, batchedID, and batchedShrink, and the inputs (executed on the GPU as well) are the black-box function $K_{blk}(\cdot)$ and batchedGen. Although Section IV and the numerical results only involve single-GPU implementation, here we add a few notes regarding the potential extension of our algorithm to multiple GPUs.

All the batched operations have a batch count set to the number of nodes at that level. The batch count decreases from $O(N)$ at the leaf level to (at most) $O(1)$ at the highest level. When the ranks have the same order of magnitudes across the levels, the computation workload per node τ remains relatively constant and hence higher parallel efficiency can be achieved for lower levels. Note that $\mathcal{H}^2$ is a $O(N)$-complexity algorithm dominated by the lower level operations, good overall parallel efficiency can be achieved for these operations. It is also worth mentioning that our batched algorithm requires only $L = O(\log N)$ kernel launches, which is a very small cost compared with the total $O(N)$ computational cost. In fact, the overhead in kernel launches is negligible compared with the total execution time, particularly for large N.

For multiple GPUs, the batch count becomes roughly the number of nodes per level divided by the number of GPUs. Moreover, all the aforementioned batched operations do not require inter-GPU communication except for batchedBSRGemm, which requires communication of the input vectors Ω. Also Line 24 of Algorithm 1 may require gathering the vectors from two GPUs into one.

V. Numerical Evaluation

In this section, we analyze the performance of Algorithm 1 on three different problems on CPUs and GPUs. We compare the memory, runtime and accuracy of our algorithm with other existing high-performance implementations of sketching-based strongly-admissible hierarchical matrix construction algorithms, including the GPU implementation of the top-down $\mathcal{H}^2$ algorithm [22] from the H2Opus library [17] and the distributed-memory CPU implementation of the top-down $\mathcal{H}$ algorithm [23] from the ButterflyPACK (v3.2.0) library [33]. To the best of our knowledge, these are the only publicly available packages supporting sketching-based construction of strongly-admissible hierarchical matrices (i.e., $\mathcal{H}^2$ or $\mathcal{H}$). Our proposed GPU implementation of Algorithm 1 and the reference algorithm in H2Opus are executed on an 80GB A100 GPU available on Perlmutter GPU nodes. Our proposed CPU implementation Algorithm 1 uses OpenBLAS[2] routines within OpenMP parallel loops for the batched operations and Thrust with the OpenMP backend for the data marshaling, which is executed using 64 OpenMP threads of an AMD EPYC 7763 processor available on Perlmutter GPU nodes. The reference algorithm in ButterflyPACK is executed on the same AMD processor using 64 MPI ranks. In addition to ButterflyPACK and H2Opus, we also consider comparison with sketching-based weakly-admissible hierarchical matrix construction algorithms implemented in STRUMPACK (v8.0.0) [34].

A. Test Problems

Throughout the paper, we consider three applications of the proposed $\mathcal{H}^2$ construction algorithm. For the first application, we look at the construction of spatial statistics covariance matrices for a 3D Gaussian spatial process on a uniform 3D distribution of points in a cube and use an exponential kernel with correlation length $l = 0.2$:

$$K(x, y) = e^{-\frac{|x-y|}{l}} \tag{8}$$

For the second application, we consider the construction of the discretized volume integral equation (IE) operator for the Helmholtz equation among a uniform 3D distribution of points in a cube and the IE operator is

$$K(x, y) = \frac{\cos(k|x-y|)}{|x-y|}, \ x \neq y \tag{9}$$

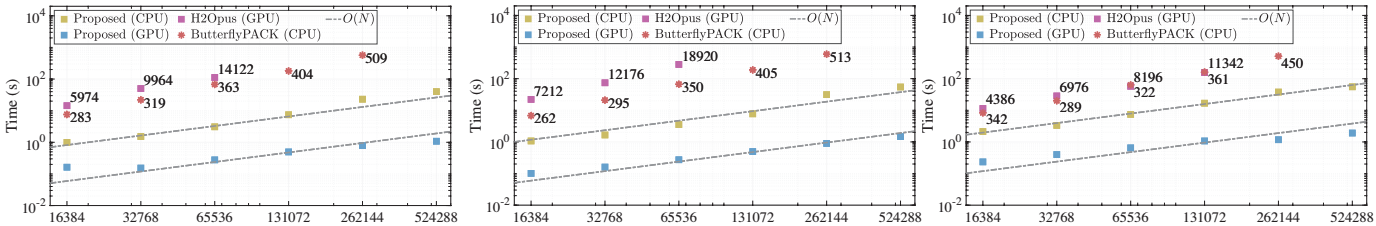
with k fixed to be 3. For these two applications, we use the fast $\mathcal{H}^2$-matrix-vector product from the H2Opus library [17] as the black box input function $K_{blk}(\cdot)$ and the direct implementation of (8) and (9) in batchedGen. For the reference CPU implementation from ButterflyPACK, we use the $\mathcal{H}$ representation for $K_{blk}(\cdot)$ and implement batchedGen on CPUs.

For the third application, we extract frontal matrices of varying sizes in full from the multifrontal factorization of a uniform-grid discretized 3D Poisson problem. We compare the performance of the proposed algorithm with other sketching-based algorithms implemented in STRUMPACK [34] (e.g. HSS [29], HODLR [22] and HODBF).

In addition to construction of the $\mathcal{H}^2$ matrix from these kernels, we also consider the updating an existing $\mathcal{H}^2$ representation of the covariance matrix with an additional low-rank product using the proposed algorithm. This is commonly encountered during the LU decomposition of hierarchical matrices or in the multifrontal factorization of sparse matrices. We use the fast $\mathcal{H}^2$-matrix-vector product from H2Opus (and fast low-rank multiplication) to perform $K_{blk}(\cdot)$, and an algorithm that extracts entries from the given $\mathcal{H}^2$ and low-rank representations to perform batchedGen.

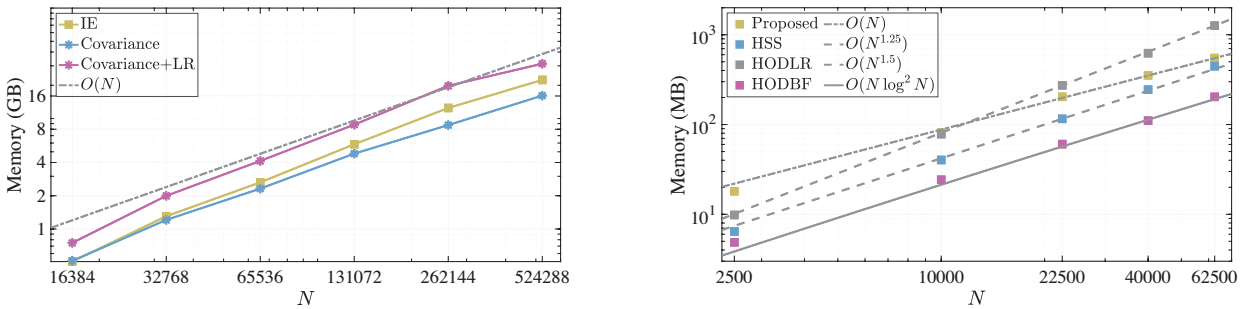
The cluster tree is constructed as a KD-tree with a leaf size of 64-256 and a dual tree traversal of the cluster tree constructs the matrix tree. We measure the approximation relative error $\frac{|K_{\text{comp}}-K|}{|K|}$ using a few iterations of the power

[2]https://github.com/OpenMathLib/OpenBLAS



(a) Covariance matrix (b) IE matrix (c) Low-rank updated covariance matrix

Fig. 5: The time of the CPU and GPU implementations of Algorithm 1 for the covariance and IE matrices as well as the covariance matrix updated with a rank 32 low-rank product. Also shown on the time plots is the top-down construction using H2Opus and ButterflyPACK with its data points labeled with the total samples taken.



(a) Memory consumption of the IE and covariance kernels (b) Memory consumption of the frontal matrices

Fig. 6: (a) The memory of Algorithm 1 for the covariance and IE matrices. (b) The memory of Algorithm 1 and a few other sketching-based algorithms in STRUMPACK for the frontal matrices.

method to approximate the 2-norm of the difference between the constructed hierarchical matrix and the provided sampler $K_{blk}(\cdot)$.

B. Computational Complexity

For the IE and covariance kernels, we examine the overall performance of the algorithm to verify the optimal complexity of the construction in time and memory. Fig. 5(a) and 5(b) show the construction time including sampling and entry generation for various covariance and IE matrices respectively on the CPU and GPU while figure 5(c) shows the same statistics for compressing the sum of $\mathcal{H}^2$ representation of the covariance matrix and a rank-32 low-rank product into a new $\mathcal{H}^2$ matrix. The memory usage of the proposed algorithm is shown in Fig. 6(a). We also show the GPU top-down construction time from the H2Opus library. The target matrices were constructed to an error threshold of 10^{-6} with an admissibility parameter $\eta = 0.7$, 256 initial samples and a leaf size of 64, while the input H2Opus matrices was constructed to a looser threshold of 10^{-5} as the implementation within H2Opus uses Cholesky QR for the orthogonalization of samples and thus it is difficult to reliably construct matrices with tight thresholds. The input ButterflyPACK matrices was also constructed with a threshold of 10^{-5}.

Our construction algorithm clearly exhibits the expected optimal runtime complexity with our GPU implementation showing speedups of up to $13\times$ over our CPU implementation, up to $660\times$ over ButterflyPACK's CPU implementation, and over $1000\times$ faster than H2Opus' GPU implementation. (see Fig. 5(a)-(c)). Note that H2Opus runs out of memory on the problems larger than 65536. Fig. 6(a) shows the expected linear growth of the memory consumption of the constructed matrices for the three test problems.

It's worth mentioning that our algorithm requires $O(1)$ (more precisely 256 for all data points in Fig. 5) number of random vectors. In stark contrast, the algorithm in ButterflyPACK [23] requires $O(\log N)$ random vectors (ranging from 262 to 513 in Fig. 5(a)-(c)). Moreover, H2Opus's implementation requires a temporary weak-admissible representation (HODLR), hence requires much more number of random vectors (up to 18920) for 3D problems, causing the code to memory crash for larger problems. In short, we remark that the proposed algorithm requires significantly less random samples particularly when the problem size N increases. This dramatic reduction in the number of random vectors, i.e., the time spent in $K_{blk}(\cdot)$, contributes most to the aforementioned speedups comparing with ButterflyPACK or H2Opus.

Also, note that the largest problem size $N = 524288$ of our GPU implementation is limited by the fact that we need to store both $K_{blk}(\cdot)$ (consisting of an existing $\mathcal{H}^2$ matrix) and the constructed $\mathcal{H}^2$ matrix on the single GPU. Given that one A100 GPU with 80GB memory is used, the $\mathcal{H}^2$ matrix can consume at most approximately 40GB memory. To handle larger problem sizes, a multi-GPU implementation will be considered as our future work.

For the frontal matrices, we only show the memory usage of

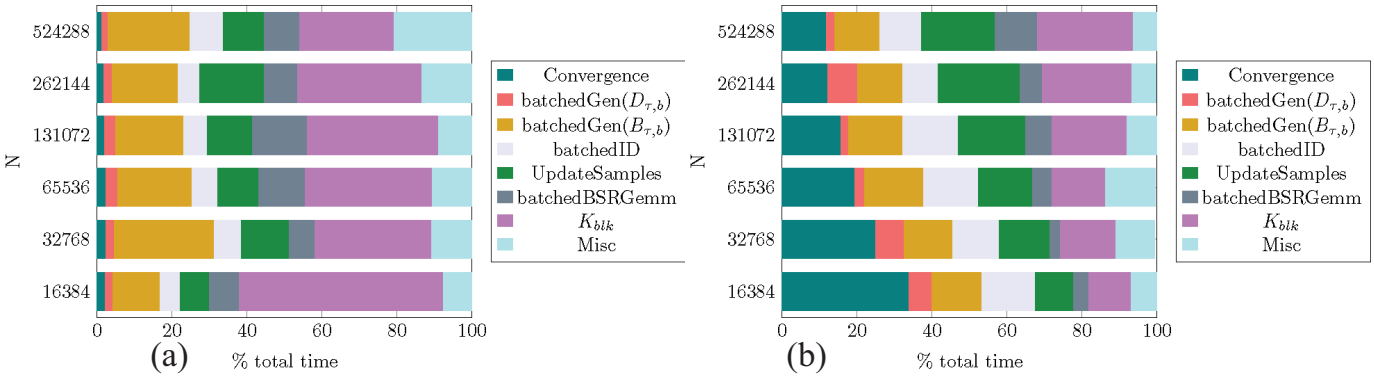


Fig. 7: A breakdown of the construction time by percentage of time taken by each phase on (a) CPU and (b) GPU for varying problem sizes of the 3D covariance matrix.

		Time	Rank range	Memory	Total samples	Sample block size	Leaf Size	Relative Error
Covariance	fixed sample	0.860	29-55	22.939	256	256	256	4.139360e-08
	adaptive	0.302	45-55	22.956	64	32	256	5.374687e-07
	fixed sample	0.348	32-55	13.906	128	128	128	8.404928e-08
	adaptive	0.246	42-55	13.930	64	32	128	5.968764e-07
IE	fixed sample	0.897	33-66	23.437	256	256	256	2.603808e-08
	adaptive	0.428	36-66	23.446	96	32	256	1.120914e-07
	fixed sample	0.517	35-66	14.998	128	128	128	6.461307e-08
	adaptive	0.392	51-67	15.081	96	32	128	1.614852e-07

TABLE II: The effect of varying the leaf size and sample block size on the memory consumption and ranks of the constructed matrix as well as runtime and approximation error with a threshold of 10^{-6} for the 3D problems of size $N = 2^{18}$.

different algorithms as the sketching operator is a full $N \times N$ matrix. This also limits the largest matrix sizes we can test. We leave the full integration of the proposed algorithm into multi-frontal sparse direct solvers as a future work. Fig. 6(b) shows the memory usage of the proposed algorithm, HSS [29], HODLR and HODBF. Clearly, our algorithm achieves the optimal $O(N)$ memory usage. However, note that the other three algorithms are weak-admissibility-based and their prefactors can be much smaller.

C. Profiling Results

Fig. 7 breaks down the runtime into the major components of the construction algorithm on the CPU and the GPU. The convergence test, where the batched QR decomposition comprises the majority of the work, represents a significantly smaller portion of the overall runtime on the CPU compared to the GPU. This is primarily caused by the batched QR implementation within KBLAS favoring larger batch sizes and smaller matrices with it's unblocked algorithm that only assigns threads to work on one column at a time. This is especially apparent for the smaller problem sizes where there isn't enough work to saturate the GPU. As the problem size increases, the overall portion of time spent in the convergence test on the GPU starts to shrink. The dense and coupling $\mathcal{H}^2$ entry generation seems to perform well on both the CPU and GPU, taking between 10-15% of the total runtime on the CPU and 15-20% on the GPU. Since we only perform the pivoted QR decompositions after we've determined that the number of samples are sufficient, the ID phases only take between 5-10% of the runtime. On both CPU and GPU, the majority of time is spent in the BSR matrix multiplication and the sampling phases, both of which are heavily matrix-matrix multiplication dependent, an operation that is particularly well suited for GPU execution. The miscellaneous section includes mostly workspace allocations which can be optimized in the future.

D. Efficacy of Adaptive Sampling

Finally, we demonstrate the effects of the adaptive sampling on a fixed 3D problem of size $N = 2^{18}$ by varying both the leaf size and the sampling block size d. Table II shows the GPU results for both the covariance and IE matrices for leaf sizes of 128 and 256 and sampling block sizes equal to the leaf size and fixed at 32. The lower leaf sizes lead to lower overall memory consumption and lower construction times while the fixed sampling block sizes lead to overall lower execution times albeit with a lower resulting accuracy and slightly higher ranks on the higher levels. This is likely due to the simple error compensation scheme not fully accounting for the approximation errors as we sweep up the tree, though the measured error is still within the threshold of 10^{-6}.

VI. Conclusion

This paper presents a GPU algorithm and implementation of a novel linear-complexity bottom-up sketching-based algorithm for constructing a $\mathcal{H}^2$ matrix. The proposed construction algorithm requires both a black-box sketching operator and an entry evaluation function, both of which are accelerated by batched GPU implementations. When applied to covariance matrices, volume IE matrices and $\mathcal{H}^2$ update operations, our proposed GPU implementation achieves up to 13× speedup

over our CPU implementation, and up to $1000\times$ speedup over an existing GPU implementation of the top-down sketching-based algorithm from the H2Opus library. Moreover, the proposed algorithm is capable of handling covariance/IE matrices with sizes up to $N = 524288$ using less than 30 GB GPU memory and we expect the algorithm can go up to $N = 1.5$ million on a single 80GB A100 GPU with further code optimizations in the future.

In addition to pushing the limit of the proposed algorithm, we also plan to investigate the GPU implementation of the inversion of the $\mathcal{H}^2$ matrix [26] and the GPU implementation of other fully sketching-based construction algorithms such as [23], [24], as well as the full integration of the proposed algorithm into sparse multifrontal solvers.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] W. Hackbusch, "A sparse matrix arithmetic based on $\mathcal{H}$-matrices. part i: Introduction to-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[2] M. Bebendorf and W. Hackbusch, "Existence of $\mathcal{H}$-matrix approximants to the inverse fe-matrix of elliptic operators with l^∞-coefficients," *Numerische Mathematik*, vol. 95, pp. 1–28, 2003.

[3] S. Börm, L. Grasedyck, and W. Hackbusch, "Introduction to hierarchical matrices with applications," *Engineering analysis with boundary elements*, vol. 27, no. 5, pp. 405–422, 2003.

[4] Y. Zhang, A. Gillman, and S. Veerapaneni, "A fast direct solver for integral equations on locally refined boundary discretizations and its application to multiphase flow simulations," *Advances in Computational Mathematics*, vol. 48, no. 5, p. 63, 2022.

[5] Y. Tu, Z. Xu, and H. Yang, "Hierarchical interpolative factorization for self green's function in 3d modified poisson-boltzmann equations," *Communications on Applied Mathematics and Computation*, pp. 1–26, 2024.

[6] J. Xia, "Randomized sparse direct solvers," *SIAM Journal on Matrix Analysis and Applications*, vol. 34, no. 1, pp. 197–227, 2013.

[7] P. Ghysels, S. L. Xiaoye, C. Gorman, and F.-H. Rouet, "A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 897–906.

[8] G. Chávez, Y. Liu, P. Ghysels, X. S. Li, and E. Rebrova, "Scalable and memory-efficient kernel ridge regression," in *2020 IEEE International parallel and distributed processing symposium (IPDPS)*. IEEE, 2020, pp. 956–965.

[9] 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.

[10] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu, "A superfast algorithm for Toeplitz systems of linear equations," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 4, pp. 1247–1266, 2008.

[11] S. Ambikasaran and E. Darve, "An $\mathcal{O}(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices: with application to radial basis function interpolation," *Journal of Scientific Computing*, vol. 57, pp. 477–501, 2013.

[12] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals, "A fast solver for HSS representations via sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 1, pp. 67–81, 2007.

[13] 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, pp. 217–247, 2012.

[14] 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.

[15] K. L. Ho and L. Ying, "Hierarchical interpolative factorization for elliptic operators: integral equations," *Communications on Pure and Applied Mathematics*, vol. 69, no. 7, pp. 1314–1353, 2016.

[16] H. Huang, X. Xing, and E. Chow, "H2pack: High-performance h2 matrix package for kernel matrices using the proxy point method," *ACM Trans. Math. Softw.*, vol. 47, no. 1, dec 2020.

[17] S. Zampini, W. Boukaram, G. Turkiyyah, O. Knio, and D. Keyes, "H2Opus: a distributed-memory multi-GPU software package for non-local operators," *Advances in Computational Mathematics*, vol. 48, no. 3, p. 31, 2022.

[18] S. Börm, "Distributed $\mathcal{H}^2$-matrices for boundary element methods," *ACM Trans. Math. Softw.*, vol. 49, no. 2, jun 2023.

[19] W. B. March, B. Xiao, C. D. Yu, and G. Biros, "ASKIT: An efficient, parallel library for high-dimensional kernel summations," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S720–S749, 2016.

[20] 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.

[21] P. Zaspel, "Algorithmic patterns for H-matrices on many-core processors," *Journal of Scientific Computing*, vol. 78, no. 2, pp. 1174–1206, 2019.

[22] 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.

[23] J. Levitt and P.-G. Martinsson, "Randomized compression of rank-structured matrices accelerated with graph coloring," *arXiv preprint arXiv:2205.03406*, 2022.

[24] A. Yesypenko and P. Martinsson, "Randomized strong recursive skeletonization: Simultaneous compression and factorization of $\mathcal{H}$-matrices in the black-box setting," *arXiv preprint arXiv:2311.01451*, 2023.

[25] 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.

[26] M. Ma and D. Jiao, "Direct solution of general $\mathcal{H}^2$ -matrices with controlled accuracy and concurrent change of cluster bases for electromagnetic analysis," *IEEE Transactions on Microwave Theory and Techniques*, vol. 67, no. 6, pp. 2114–2127, 2019.

[27] T. Liang, C. Chen, P.-G. Martinsson, and G. Biros, "An $O(N)$ distributed-memory parallel direct solver for planar integral equations," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 440–452.

[28] Q. Ma, S. Deshmukh, and R. Yokota, "Scalable linear time dense direct solver for 3-d problems without trailing sub-matrix dependencies," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–12.

[29] 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.

[30] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[31] A. Abdelfattah, D. Keyes, and H. Ltaief, "KBLAS: An optimized library for dense matrix-vector multiplication on GPU accelerators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 3, pp. 1–31, 2016.

[32] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "Framework for batched and GPU-resident factorization algorithms to block Householder transformations," in *ISC High Performance*, Springer. Frankfurt, Germany: Springer, 07-2015 2015.

[33] Y. Liu and USDOE, "ButterflyPACK," 11 2018. [Online]. Available: https://www.osti.gov//servlets/purl/1564244

[34] P. Ghysels and USDOE, "STRUMPACK – STRUctured Matrices PACKage," 12 2014. [Online]. Available: https://www.osti.gov/biblio/1328126