Geometry-Oblivious FMM for Compressing Dense SPD Matrices

Chenhan D. Yu
Department of Computer Science
†Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX
chenhan@cs.utexas.edu

Severin Reiz
Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX
s.reiz@tum.de

James Levitt
Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX
jlevitt@ices.utexas.edu

George Biros
Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX
gbiros@acm.org

ABSTRACT

We present GOFMM (geometry-oblivious FMM), a novel method that creates a hierarchical low-rank approximation, or "compression," of an arbitrary dense symmetric positive definite (SPD) matrix. For many applications, GOFMM enables an approximate matrix-vector multiplication in \( N \log N \) or even \( N \) time, where \( N \) is the matrix size. Compression requires \( N \log N \) storage and work. In general, our scheme belongs to the family of hierarchical matrix approximation methods. In particular, it generalizes the fast multipole method (FMM) to a purely algebraic setting by only requiring the ability to sample matrix entries. Neither geometric information (i.e., point coordinates) nor knowledge of how the matrix entries have been generated is required, thus the term "geometry-oblivious." Also, we introduce a shared-memory parallel scheme for hierarchical matrix computations that reduces synchronization barriers. We present results on the Intel Knights Landing and Haswell architectures, and on the NVIDIA Pascal architecture for a variety of matrices.

CCS CONCEPTS
• Theory of computation → Numeric approximation algorithms; Sketching and sampling; • Mathematics of computing → Mathematical software performance; Kernel density estimators; • Computing methodologies → Linear algebra algorithms; Parallel algorithms; Kernel methods; • Computer systems organization → Multicore architectures; Heterogeneous (hybrid) systems;

KEYWORDS
Geometry-oblivious, Fast Multipole Methods, Hierarchical Matrices, Fast Matrix Multiplication, Heterogeneous Computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference format:
Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. 2017. Geometry-Oblivious FMM for Compressing Dense SPD Matrices. In Proceedings of SC17, Denver, CO, USA, November 12–17, 2017, 14 pages.
DOI: 10.1145/3126908.3126921

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC17, Denver, CO, USA
© 2017 ACM. 978-1-4503-5114-0/17/11…$15.00
DOI: 10.1145/3126908.3126921

1 INTRODUCTION

We present GOFMM, a novel algorithm for the approximation of dense symmetric positive definite (SPD) matrices. GOFMM can be used for compressing a dense matrix and accelerating matrix-vector multiplication operations. As an example, in Figure 1 we report timings for an SGEMM (single-precision matrix-matrix multiplication) operation using an optimized dense matrix library and compare with the GOFMM-compressed version.

Problem statement: Let \( K \in \mathbb{R}^{N \times N} \) be a dense SPD matrix, with \( K = K^T \) and \( x^T K x > 0, \forall x \in \mathbb{R}^N, x \neq 0 \). Since \( K \) is dense, it requires \( O(N^2) \) storage and \( O(N^2) \) work for a matrix-vector multiplication (hereby "matvec"). Assuming the evaluation of a single matrix entry \( K_{ij} \) requires \( O(1) \) work, we wish to construct a hierarchical matrix \( \tilde{K} \) with the following properties: (1) constructing \( \tilde{K} \) requires \( O(N \log N) \) work; (2) a matvec with \( \tilde{K} \) also requires \( O(N \log N) \) work; and (3) \( \| K - \tilde{K} \| \leq \varepsilon \| K \| \), where \( 0 < \varepsilon < 1 \) is a user-defined error tolerance. In other words, given any SPD matrix \( K \), our task is to construct a hierarchically low-rank matrix compression \( \tilde{K} \) such that the relative error \( \| K - \tilde{K} \| / \| K \| \) is small.
The only required input to our algorithm is a routine that returns a submatrix \( K_{ij} \), for arbitrary row and column index sets \( I \) and \( J \). For certain matrices, we can achieve these goals with GOFMM. Our scheme belongs to the class of hierarchical matrix approximation methods. The constant in the complexity estimate depends on the user-defined error tolerance, the structure of the underlying matrix, and the GOFMM variant. Let us remark and emphasize that our approximation scheme cannot guarantee both accuracy and work complexity simultaneously, since an arbitrary SPD matrix may not admit a good hierarchical low-rank matrix approximation (see §2).

We say that a matrix \( \tilde{K} \) has a hierarchically low-rank structure, i.e., \( \tilde{K} \) is an \( \mathcal{H} \)-matrix [4, 21], if
\[
\tilde{K} = D + S + UV,
\]
where \( D \) is block-diagonal with every block being an \( \mathcal{H} \)-matrix, \( U \) and \( V \) are low rank, and \( S \) is sparse. At the base case of this recursive definition, the blocks of \( D \) are dense small matrices. An \( \mathcal{H} \)-matrix matvec requires \( O(N\log N) \) work, and the constant in the complexity estimate depends on the rank of \( U \) and \( V \). Depending on the construction algorithm, this complexity can go down to \( O(N) \). Although such matrices are rare in real-world applications, it is quite common to find matrices that can be approximated arbitrarily well by an \( \mathcal{H} \)-matrix.

One important observation is that this hierarchical low-rank structure is not invariant to row and column permutations. Therefore any algorithm for constructing \( \tilde{K} \) must first appropriately permute \( K \) before constructing the matrices \( U, V, D, \) and \( S \). Existing algorithms rely on the matrix entries \( K_{ij} \) being “interactions” (pairwise functions) between points \( \{x_i\}_{i=1}^N \in \mathbb{R}^d \) and permute \( K \) either by clustering the points (typically using some tree data-structure) or by using graph partitioning techniques (if \( K \) is sparse). GOFMM requires neither geometric information nor sparsity.

**Background and significance.** Dense SPD matrices appear in scientific computing, statistical inference, and data analytics. They appear in Cholesky and LU factorization [17], in Schur complement matrices for saddle point problems [6], in Hessian operators in optimization [36], in kernel methods for statistical learning [18, 24], and in N-body methods and integral equations [20, 21]. In many applications, the entries of the input matrix \( K \) are given by \( K_{ij} = \mathcal{K}(x_i, x_j) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \), where \( \mathcal{K} \) is a kernel function. Examples of kernel functions are radial basis functions, Green’s functions, and angle similarity functions. For such kernel matrices, the input is not a matrix, but only the points \( \{x_i\}_{i=1}^N \). The points are used to appropriately permute the matrix using spatial data structures. Furthermore, the construction of the sparse correction \( S \) uses nearest-neighbor structure of the input points. The low-rank matrices \( U, V \) can be either analytically computed using evaluations of the kernel function, or semi-algebraically computed using fictitious points (or equivalent points), or using algebraic sampling-based methods that use geometric information. In a nutshell, geometric information is used in all aspects of an \( \mathcal{H} \)-matrix method.

In many cases however, such points and kernel functions are not available. For example, in dense graphs in data analysis (e.g., social networks, protein interactions). Related matrices include graph Laplacian operators and their inverses. Additional examples include frontal matrices and Schur complements in factorization of sparse matrices; Hessian operators in optimization; and kernel methods in machine learning without points (e.g., word sequences and diffusion on graphs [7, 26]).

**Contributions.** GOFMM is inspired by the rich literature of algorithms for matrix sketching, hierarchical matrices, and fast multipole methods. Its unique feature is that by using only matrix evaluations it generalizes FMM ideas to compressing arbitrary SPD matrices. In more detail, our contributions are summarized below.

- A result from reproducing kernel Hilbert space theory is that any SPD matrix corresponds to a Gram matrix of vectors in some, unknown Gram (or feature) space [24]. Based on this result, the matrix entries are inner products, which we use to define distances. These distances allow us to design an efficient, purely algebraic FMM method.
- The key algorithmic components of GOFMM (and other hierarchical matrix and FMM codes) are tree traversals. We test parallel level-by-level traversals, out-of-order traversals using OpenMP’s advanced task scheduling and an in-house tree-task scheduler. We found that scheduling significantly improves the performance when compared to level-by-level tree traversals. We also use this scheduling to support heterogeneous architectures.
- We conduct extensive experiments to demonstrate the feasibility of the proposed approach. We test our code on 22 different matrices related to machine learning, stencil PDEs, spectral PDEs, inverse problems, and graph Laplacian operators. We perform numerical experiments on Intel Haswell and KNL, Qualcomm ARM, and NVIDIA Pascal architectures. Finally, we compare with three state-of-the-art codes: HODLR, STRUMPACK, and ASKIT.

GOFMM also has several additional capabilities. If points and kernel functions (or Green’s function) are available, they can be utilized in a similar way to the algebraic FMM code ASKIT we previously developed [30, 32]. GOFMM currently supports three different measures of distance: geometric point-based (if available), Gram-space \( \ell^2 \) distance, and Gram-space angle distance. GOFMM has support for matrices with multiple right hand sides, which is useful for Monte-Carlo sampling, optimization, and blocked Krylov methods.

**Limitations.** GOFMM is restricted to SPD matrices. (However, if we are given points, the method becomes similar to existing methods.) GOFMM guarantees symmetry of \( \tilde{K} \), but if \( \|K - \tilde{K}\|/\|K\| \) is large, positive definiteness may be compromised. To reiterate, GOFMM cannot simultaneously guarantee both accuracy and work complexity. This initial implementation of GOFMM supports shared-memory parallelism and accelerators, but not distributed memory architectures. The current version of GOFMM also has several parameters that require manual tuning. Often, the main goal of building \( \mathcal{H} \)-matrix approximations is to construct a factorization of \( K \), a topic we do not discuss in this paper. Our method requires the ability to evaluate matrix entries and the complexity estimates require that these entries can be computed in \( O(1) \) time. If \( K \) is only available through matrix-free interfaces, these assumptions may not be satisfied. Other algorithms, like STRUMPACK, have inherent support for such matrix-free compression.

**Related work.** The literature on hierarchical matrix methods and fast multipole methods is vast. Our discussion is brief and limited to the most related work.
Low-rank approximations. The most popular approach for compressing arbitrary matrices is a global low-rank approximation using randomized linear algebra. In (1), this is equivalent to setting \( D \) and \( S \) to zero and constructing only \( U \) and \( V \). Examples include the CUR [29] factorization, the Nyström approximation [42], the adaptive cross approximation [5], and randomized rank-revealing factorizations [22, 36]. These techniques can also be used for \( \mathcal{H} \)-matrix approximations when \( D \) is not zero. Instead of applying them to \( K \), we can apply them to the off-diagonal blocks of \( K \). FMM-specific techniques that are a mix between analytic and algebraic methods include kernel-independent methods [35, 45] and the black-box FMM [15]. Constructing both \( U \) and \( V \) accurately and with optimal complexity is hard. The most robust algorithms require \( O(N^2) \) complexity or higher (randomized methods and leverage-score sampling) since they require one to “touch” all the entries of the matrix (or block) to be approximated.

Permuting the matrix. When \( K \) is sparse, the method of choice uses graph-partitioning. This doesn’t scale to dense matrices because practical graph partitioning algorithms scale at least linearly with the number of edges and thus the construction cost would be at least \( O(N^2) \) [1, 25].

\( \mathcal{H} \)-matrix methods and software. Treecodes and fast multipole methods originally were developed for N-body problems and integral equations. Algebraic variants led the way to the abstraction of \( \mathcal{H} \)-matrix methods and the application to the factorization of sparse systems arising from the discretization of elliptic PDEs [2, 4, 19, 21, 23, 43].

Let us briefly summarize the \( \mathcal{H} \)-matrix classification. Recall the decomposition \( K = D + S + UV \), (1). If \( S \) is zero the approximation is called a hierarchically off-diagonal low rank (HODLR) scheme. In addition to \( S \) being zero, if the \( \mathcal{H} \)-matrix decomposition of \( D \) is used to construct \( U \), \( V \) we have a hierarchically semi-separable (HSS) scheme. If \( S \) is not zero we have a generic \( \mathcal{H} \)-matrix; but if the \( U, V \) terms are constructed in a nested way then we have an \( \mathcal{H}^2 \)-matrix or an FMM depending on more technical details. HSS and HODLR matrices lead to very efficient approximation algorithms for \( K^{-1} \). However, \( \mathcal{H}^2 \) and FMM compression schemes better control the maximum rank of the \( U \) and \( V \) matrices than HODLR and HSS schemes. For the latter, the rank of \( U \) and \( V \) can grow with \( N \) [8] and the complexity bounds are no longer valid. Recently, here have been algorithms to effectively compress FMM and \( \mathcal{H}^2 \)-matrices [12, 46]. One of the most scalable methods is STRUMPACK [16, 34, 38], which constructs an HSS approximation of a square matrix (not necessarily SPD) and then uses it to construct an approximate factorization. For dense matrices STRUMPACK uses the lexicographic ordering. If no fast matrix-vector multiplication is available, STRUMPACK requires \( O(N^2) \) work for compressing a dense SPD matrix, and \( O(N) \) work for the matvec.

### 2 METHODS

Given \( K \in \mathbb{R}^{N \times N} \), GOFMM aims to construct an \( \mathcal{H} \)-matrix \( \widetilde{K} \) in the form of (1) such that we can approximate

\[
u = Kw = \widetilde{K}w, \quad \text{for } w \in \mathbb{R}^N.
\]

When points \( \{x_i\}_{i=1}^N \) are available such that \( K_{ij} = K(x_i, x_j) \), the recursive partitioning on \( D \) and the low-rank structure \( U V \) use distances between \( x_i \) and \( x_j \). Existing FMM methods approximate \( K_{ij} \) when \( x_i \) and \( x_j \) are sufficiently far from each other. Otherwise, \( K_{ij} \) is not approximated and it is placed either in \( D \) or in \( S \). We call this distance-based criterion near-far pruning.

To define such a pruning scheme without \( \{x_i\}_{i=1}^N \), we need a notion of distance between two matrix indices \( i \) and \( j \). We define such a distance in the next section. With it, we can permute \( K \) and define neighbors for each index \( i \). In §2.2, we describe a task-based algebraic FMM that only relies on the distance we define. Finally in §2.3, we discuss task parallelism and scheduling.

### 2.1 Geometry-oblivious techniques

In this section, we introduce the machinery for using GOFMM in a geometry-oblivious manner. Throughout the following discussion, we refer to a set of indices \( I = \{1, \ldots, N\} \), where index \( i \) corresponds to the \( i \)th row (or column) of the matrix \( K \) in the original ordering. Our objective is to find a permutation of \( I \) so that \( K \) can be approximated by an \( \mathcal{H} \)-matrix. The key is to define a distance between a pair of indices \( i, j \in I \), denoted as \( d_{ij} \). Using the distances, we then perform a hierarchical clustering of \( I \), which is used to define the permutation and determine which interactions go into the sparse correction \( S \) (using nearest neighbors).

We define three measures of distance including the point-based Euclidean distance (if data points are available), a Gram-space Euclidean distance, and a Gram-space angle distance.

**Geometric-\( l^2 \)**. If we are given points \( \{x_i\}_{i=1}^N \), then \( d_{ij} = ||x_i - x_j||_2 \) is the geometric \( l^2 \) distance. This will be the geometry-aware reference implementation for cases where points are given.

**Gram-\( l^2 \) (or “kernel” distance)**. Since \( K \) is SPD, it is the Gram matrix of some set of unknown Gram vectors, \( \{\phi_i\}_{i=1}^N \subset \mathbb{R}^N \) [39], proposition 2.16, page 44). That is, \( K_{ij} = \langle \phi_i, \phi_j \rangle \), where \( \langle \cdot, \cdot \rangle \) denotes the \( l^2 \) inner product in \( \mathbb{R}^N \). We define the Gram \( l^2 \) distance as \( d_{ij} = ||\phi_i - \phi_j||_2 \). Computing the kernel distance only requires

| METHOD | MATRIX | LOW-RANK | PERM | S |
|--------|--------|----------|------|---|
| FMM [10] | \( K(x_i, x_j) \) | EXP | OCTREE | Y |
| KIFMM [45] | \( K(x_i, x_j) \) | EQU | OCTREE | Y |
| BBFMM [15] | \( K(x_i, x_j) \) | EQU | OCTREE | Y |
| HODLR [3] |  | ALG | NONE | N |
| STRUMPACK [38] |  | ALG | NONE | N |
| ASKIT [33] | \( K(x_i, x_j) \) | ALG | TREE | Y |
| MLPACK [13] | \( K(x_i, x_j) \) | EQU | TREE | Y |
| GOFMM | \( K_{ij} \) | ALG | TREE | Y |

Table 1: We summarize the main features of different \( \mathcal{H} \)-matrix methods/odes for dense matrices. “MATRIX” indicates whether the method requires a kernel function and points—indicated by \( K(x_i, x_j) \)—or it just requires kernel entries—indicated by \( K_{ij} \). "LOW-RANK" indicates the method used for the off-diagonal low-rank approximations: "EXP" indicates kernel function-dependent analytic expansions; "EQU" indicates the use of equivalent points (restricted to low d problems); "ALG" indicates an algebraic method. "PERM" indicates the permutation scheme used for dense matrices: "OCTREE" indicates that the scheme doesn’t generalize to high dimensions; "NONE" indicates that the input lexicographic order is used; and "TREE" indicates geometric partitioning that scales to high dimensions. \( S \) indicates whether a sparse correction (FMM or \( \mathcal{H} \)-) is supported. In §4, we present comparisons with ASKIT, STRUMPACK, and HODLR.
Algorithm 2.1 \([1, r] = \text{metricSplit}(\alpha)\)

\[
p = \text{argmax}\{(d_{pq}|i \in \alpha)\};\quad q = \text{argmax}\{(d_{ip}|i \in \alpha)\};
[1, r] = \text{medianSplit}((d_{ip} - d_{iq}|i \in \alpha));
\]

three entries of \(K\):

\[
d_{ij}^2 = \|\phi_i\|^2 + \|\phi_j\|^2 - 2(\phi_i, \phi_j) = K_{ii} + K_{jj} - 2K_{ij}. \tag{3}\]

Gram angles (or "angle" distance). Our third measure of distance considers angles between Gram vectors, which is based on the standard sine distance (cosine similarity) in inner product spaces. We define the Gram angle distance as \(d_{ij} = \sin^2 \left(\angle(\phi_i, \phi_j)\right)\) \([-1, 1]\). This expression is chosen so that \(d_{ij}\) is small for nearly collinear Gram vectors, large for nearly orthogonal Gram vectors, and \(d_{ij}\) is inexpensive to compute. Although the value \(d_{ij}\) may seem arbitrary, we only compare values for the purpose of ordering, so any equivalent metric will do. Computing an angle distance only requires three entries of \(K\):

\[
d_{ij} = 1 - \cos^2 \left(\angle(\phi_i, \phi_j)\right) = 1 - K_{ij}^2/(K_{ii}K_{jj}). \tag{4}\]

To reiterate for emphasis, \(d_{ij}\) define proper distances (metrics) because \(K\) is SPD. And with distances, we can apply FMM.

Tree partitioning and nearest neighbor searches. \(K\) is permuted using a balanced binary tree. The root node is assigned with the full set of points, and the tree is constructed recursively by splitting a node’s points evenly between two child nodes according to the pairwise distance metric \(d_{ij}\). The splitting terminates at nodes with some pre-determined leaf size \(m\). The leaf nodes then define a partial ordering of the indices: if leaf \(\alpha\) is anywhere to the left of leaf \(\beta\), then the indices of \(\alpha\) precede those of \(\beta\). We use this ordering to permute rows and columns of \(K\). In the remainder of this paper, we use the notation \(\alpha, \beta\) to refer interchangeably to a node or the set of indices belonging to the node.

In our implementation, we use a metric ball tree [33], which splits data points according to their pairwise distances. For geometric distances, the tree construction costs \(O(N \log N)\). But Gram distances (kernel and angle) require sampling to avoid \(O(N^2)\) costs. Suppose we use one of the Gram distances to split an interior node \(\alpha\) between its left child \(1\) and right child \(r\). We define \(c = \frac{1}{n_c} \sum \phi_i\) to be an approximate centroid\(^1\) taken over a small sample of \(n_c\) Gram vectors belonging to \(\alpha\). \(n_c\) is \(O(1)\). Next, we find the point \(p\) that is farthest away in distance from \(c\), and the point \(q\) that is farthest away from \(p\). Then we split the indices \(i \in \alpha\) on the values \(d_{ip} - d_{iq}\), which measures the degree to which \(i\) is closer to \(p\) than to \(q\). This approach is outlined in Algorithm 2.1.

We perform all nearest neighbors (ANN) search using randomized trees that are constructed in exactly the same way as the metric partitioning tree, except that \(p\) and \(q\) are chosen randomly. The search algorithm is described in [44] and (briefly) in the next section.

2.2 Algebraic Fast Multipole Method

\(H\)-matrix methods (including algebraic FMM) have two phases: compression and evaluation. As we discussed in the introduction,

\[K_{\alpha\alpha} = \begin{bmatrix} K_{11} & 0 & 0 \\ 0 & S_{1r} & 0 \\ 0 & 0 & U_{V_{1r}} \end{bmatrix} \tag{5}\]

where \(1\) and \(r\) are left and right child of the treenode \(\alpha\). Each node \(\alpha\) contains a set of matrix indices and the two children evenly split the indices such that \(\alpha = 1 \cup r\). (We overload the notation \(\alpha, \beta, 1\) and \(r\) to denote the matrix indices that those treenode own.) In Figure 2, the blue blocks depict \(S\) (at all levels) and \(D\) (in the leaf level), and the pink blocks depict the \(UV\) matrices.

We use four tree traversals to describe the algorithms in GOFMM: postorder (POST), preorder (PRE), any order (ANY), and any order-leafs only (LEAF). By "task" we refer to a computation that occurs when we visit a tree node during a traversal. We list all tasks required by the compression phase (Algorithm 2.2) and evaluation phase (Algorithm 2.7) in Table 2.

GOFMM compression starts by creating the binary metric ball tree in Algorithm 2.2 that represents the binary partitioning (and encodes a symmetric permutation of matrix \(K\)). This requires the distance metric \(d_{ij}\) and a preorder traversal (PRE) of the first task SPLI(\(\alpha\)) in Table 2.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Task & Operations & FLOPS \\
\hline
SPLI(\(\alpha\)) & split \(\alpha\) into 1 and \(r\) Algorithm 2.1 & \(|r|\) \\
\hline
ANN(\(\alpha\)) & update \(N_{\alpha}\) with KNN(\(K_{\alpha\alpha}\)) & \(m^2\) \\
\hline
SCEL(\(\alpha\)) & in Algorithm 2.6 & \(2s^2 + 2m^3\) \\
\hline
COEF(\(\alpha\)) & \(P_{\alpha\alpha} = P_{\alpha[r]}\) in Algorithm 2.6 & \(s^3\) \\
\hline
N2S(\(\alpha\)) & if \(\alpha\) is leaf then \(w_{\alpha} = P_{\alpha[r]} w_{\alpha}\) else \(w_{\alpha} = P_{\alpha[r]}[w_{\alpha}]\{\hat{w}_{\alpha}\} \) & \(2msr\) \\
\hline
SKba(\(\beta\)) & \(\forall \alpha \in Far(\beta), K_{\beta\alpha} = K(\beta, \alpha)\) & \(ds^2|\text{Far}(\beta)|\) \\
\hline
S2S(\(\beta\)) & \(\hat{u}_{\beta} = \sum_{\alpha \in \text{Far}(\beta)} K_{\beta\alpha} w_{\alpha}\) else \(\{u_{\beta}; \hat{u}_{\beta}\} = P_{\beta[r]} \{u_{\beta}; \hat{u}_{\beta}\} \) & \(2s^2r|\text{Far}(\beta)|\) \\
\hline
S2N(\(\beta\)) & if \(\alpha\) is leaf then \(u_{\beta} = P_{\beta[r]} \{u_{\beta}; \hat{u}_{\beta}\} \) else \(\{u_{\beta}; \hat{u}_{\beta}\} = P_{\beta[r]} \{u_{\beta}; \hat{u}_{\beta}\} \) & \(2s^2r|\text{Far}(\beta)|\) \\
\hline
Kba(\(\beta\)) & \(\forall \alpha \in \text{Near}(\beta), K_{\beta\alpha} = K(\beta, \alpha)\) & \(m^2|\text{Near}(\beta)|\) \\
\hline
L2L(\(\beta\)) & \(u_{\beta} = \sum_{\alpha \in \text{Near}(\beta)} K_{\beta\alpha} w_{\alpha}\) & \(2m^2s^3|\text{Near}(\beta)|\) \\
\hline
\end{tabular}
\caption{Tasks and their costs in FLOPS. SPLI (tree splitting), ANN (all nearest-neighbors), SCEL (skeltonization), COEF (interpolation) Skba and Kba (caching submatrices) occur in the compression phase. Interactions N2S (nodes to skeletons), S2S (skeletons to skeletons), S2N (skeletons to nodes), and L2L (leaves to leaves) occur in the evaluation phase.}
\end{table}

\(^1\)Computing the true centroid over all data points would result in \(O(N^2)\) work.
by merging all neighbors of \(i \in \alpha\). For non-leaf nodes the list is constructed recursively \cite{30}.

In each iteration, we create a randomized projection tree \cite{14,27,33}, and we search for neighbors of \(i\) only in the leaf node \(\alpha\) that contains \(i\) using an exhaustive search \cite{47}. That is, for each \(i \in \alpha\), we only search for small \(d_{ij}\) where \(j \in \alpha\) as well. Due to the randomness, in each iteration leaf node \(\alpha\) may be assigned with a different partition, which gradually cover all neighbors during the local exhaustive search. To get a set of approximate neighbors, the iteration stops after reaching 80% accuracy or 10 iterations.

80% and 10 iterations are chosen empirically. In our experiments, we found that more accurate nearest neighbors do not improve the GOFMM approximation. More specifically, nearest-neighbors provide a guess of important matrix entries, which are used in importance sampling and selecting near interactions. In our previous work \cite{33}, we show that whether neighbors can improve the accuracy depends on the data (points or Gram vectors) in different scales. Typically, neighbor-pruning works better if the intrinsic dimensionality of the data is low. Consequently, random projection based ANN methods also converge faster \cite{44}. Otherwise, it is likely that neighbors will not improve the accuracy too much. Typically even smaller values are sufficient. We use 80% to be conservative.

Near list of a node \(\alpha\). Leaf nodes \(\alpha, \beta\) are considered near if \(\alpha \cap N(\beta) \neq \emptyset\) using Morton ID \(i\). For each leaf node \(\alpha\), \(\beta\) is constructed using LeafNear (Algorithm 2.3). For each neighbor \(i \in N(\beta)\), LeafNear \(\beta\) adds MortonID \(i\) to \(\beta\). Notice that the size of \(N(\beta)\) determines the number of direct evaluations (blue blocks in Figure 2) in the off-diagonal blocks. To prevent the cost from growing too fast, we introduce a user-defined parameter budget such that

\[
|N(\beta)| < \text{budget} \times (N/m).
\]  \hfill (6)

While looping over neighbor \(i \in N(\beta)\), instead of directly adding MortonID \(i\) to \(N(\beta)\), we only mark it with a ballot. Then we insert candidates to \(N(\beta)\) according to their votes until \(6\) is reached. To enforce symmetry of \(K\), we loop over all \(N\) lists and enforce the following: if \(\alpha \in N(\beta)\) then \(\beta \in N(\alpha)\).

Far list of a node \(\alpha\). Far \(\alpha\) is constructed in two steps in Algorithm 2.2, representing submatrices in the off-diagonal blocks that can be approximated. First for each leaf node \(\beta\), we invoke FindFar \(\beta, \text{root}\) (Algorithm 2.4). Upon visiting \(\alpha\), we check whether \(\alpha\) is a parent of any leaf node in \(N(\beta)\) using MortonID. If so, we recur to the two children of \(\alpha\); otherwise, we add \(\alpha\) to \(\text{Far}(\beta)\) (i.e., \(K_{\beta\alpha}\) can be approximated). The second step is a postorder traversal on MergeFar(\(\text{root}\)) (Algorithm 2.5). This process merges the common nodes from two children lists \(\text{Far}(i)\) and \(\text{Far}(\tau)\) to create larger off-diagonal blocks for approximation. These common nodes are removed from the children and added to their parent list \(\text{Far}(\alpha)\). In Figure 2, FindFar can be identified by the smallest square pink blocks, and MergeFar merges small pink blocks into larger blocks.

Low-rank approximation. We approximate off-diagonal matrix blocks with a nested interpolative decomposition (ID) \cite{22}. Let \(\beta\) be the indices in a leaf node and \(l = \{1, \ldots, N\} \setminus \beta\) be the set complement. The skeletonization of \(\beta\) is a rank-s approximation of its
off-diagonal blocks $K_{Iβ}$ using the ID, which we write as
\[ K_{Iβ} ≈ K_{Iβ}^p β^p \] (7)
where $β ⊂ β$ is the skeleton of $β$, $K_{Iβ}^p ∈ \mathbb{R}^{(N−|β|) × s}$ is a column submatrix of $K_{Iβ}$, and $P_{ββ}^p ∈ \mathbb{R}^{s × |β|}$ is a matrix of interpolation coefficients, where $s$ is the approximation rank.

To efficiently compute this approximation, we select a sample subset $I' ⊂ I$ using neighbor-based importance sampling [33]. We then perform a rank-revealing QR factorization (GEQP3) on $K_{Iβ}^p$. The skeletons $β$ are selected to be the first $s$ pivots, and the matrix $P_{ββ}^p$ is computed by a triangular solve (TRSM) using the triangular factor $R$. The rank $s$ is chosen adaptively such that $s_{α+1}(K_{Iβ}^p) < τ$, where $s_{α+1}(K_{Iβ}^p)$ is the estimated $s + 1$ singular value and $τ$ is related to a user-specified error tolerance.

For an internal node $α$, we form the skeletonization in the same way, except that the columns are also sampled using the skeletons of the children of $α$. That is, the ID is computed for $K_{I[I]}$: $K_{I[I]} ≈ K_{Iα}^p α^p[I]$. (8)

This way, the skeletons are nested: $α ⊂ I ⊂ I^r$ contains the skeletons of the children of $α$.

Algorithm 2.6

\[ \textbf{Algorithm 2.6} \quad \{α, P_{αα}\} = \text{Skeleton}(α) \]

if $α$ is leaf then return $\{α, P_{αα}\} = \text{ID}(α)$; 
[{$I$, $R$} = Skeleton($I$); $[\tilde{I}, \tilde{r}] = \text{Skeleton}(r)$; return $\{\tilde{α}, P_{\tilde{α}\tilde{α}}[1]\} = \text{ID}(\tilde{I}[1])$;

We never explicitly form $\alpha_i$ with a postorder traversal. There are two tasks for each tree node $α$ listed in Table 2: (1) SKEL ($α$) selects $α$ (in the critical path) and (2) COEF ($α$) computes $P_{α[I]}$. Notice that in Algorithm 2.2 only SKEL ($α$) needs to be executed in postorder (POST), but COEF ($α$) can be in any order (ANY) as long as SKEL ($α$) is finished. Such parallelism can only be specified at the task level, which later inspires our task-based parallelism in §2.3. At the end of the compression, we can optionally evaluate and cache all $K_{βα}$ in $\text{Near}(β)$ and all $K_{βα}$ in $\text{Far}(β)$ by executing $\text{Eval}(β)$ and $\text{SKEval}(β)$ in any order. Given enough memory (at least $O(N)$ for all $K_{βα}$), caching can reduce the time spent on evaluating and gathering submatrices.

Evaluation. Following Algorithm 2.7 a four-step process for computing (2). The idea is to approximate each $\text{matvec} u_β ≈ K_{βα} w_α$ in $\text{Far}(β)$ using a two-sided ID to accumulate $P_{ββ}^p K_{βα}^p P_{αα}^p w_α$, where $P_{αα}, P_{ββ}$ are given by the telescoping expression (10). For more details, see [30].

The skeletons $\hat{α}$ is computed by a triangular solve (TRSM) using the triangular factor $R$. To efficiently compute this approximation, we select a sample subset $I' ⊂ I$ using neighbor-based importance sampling [33]. We then perform a rank-revealing QR factorization (GEQP3) on $K_{Iβ}^p$. The skeletons $β$ are selected to be the first $s$ pivots, and the matrix $P_{ββ}^p$ is computed by a triangular solve (TRSM) using the triangular factor $R$. The rank $s$ is chosen adaptively such that $s_{α+1}(K_{Iβ}^p) < τ$, where $s_{α+1}(K_{Iβ}^p)$ is the estimated $s + 1$ singular value and $τ$ is related to a user-specified error tolerance.

For an internal node $α$, we form the skeletonization in the same way, except that the columns are also sampled using the skeletons of the children of $α$. That is, the ID is computed for $K_{I[I]}$: $K_{I[I]} ≈ K_{Iα}^p α^p[I]$. (8)

This way, the skeletons are nested: $α ⊂ I ⊂ I^r$ contains the skeletons of the children of $α$.

Algorithm 2.7

\[ \textbf{Algorithm 2.7} \quad \text{Evaluate}(u, w) \]

1. (POST) $N2S(α)$ # compute skeleton weights $\tilde{w}$
2. (ANY) $S2S(β)$ # apply skeleton basis $K_{βα}$
3. (PRE) $S2N(β)$ # accumulate skeleton potentials $\tilde{u}$
4. (ANY) $L2L(β)$ # accumulate direct $\text{matvec}$ to $u$

The first step is to perform a postorder traversal (POST) on $N2S(α)$ (Nodes To Skeletons). This computes the skeleton weights $\tilde{w}_α = P_{αα}^p w_α$ for each leaf node, and $\tilde{w}_α = P_{α[I]}[\tilde{w}_1; \tilde{w}_r]$ for each inner node. Recall that in COEF ($α$), we have computed $P_{αα}^p$ for each leaf node and $P_{α[I]}$ for each internal node. $S2S(β)$ (Skeletons To Skeletons) applies the skeleton basis $K_{βα}$ and accumulates skeleton potentials $\tilde{u}$ for each node: $\tilde{u}_β = \sum_{α ∈ \text{Far}(β)} K_{βα}^p \tilde{w}_α$. As soon as $\tilde{w}_α$ are computed in $N2S$, $S2S$ can be executed in any order. $S2N(β)$ (Skeletons To Nodes) performs interpolation on the left and accumulates $\tilde{u}$ with a preorder traversal. This uses the transpose of (10). For each node $β$, we accumulate $[\tilde{u}_1; \tilde{u}_r] = P_{ββ}^p \tilde{u}_β$ to its children. In the leaf node, $u_β = P_{ββ}^p \tilde{u}_β$ directly accumulates to the output. These three tasks compute all $\text{matvec}$ for the far nodes (pink blocks in Figure 2). All $\text{matvec}$ on $K_{βα}$ in $\text{Near}(β)$ (blue blocks) are computed by $L2L(β)$ (Leaves To Leaves) and directly accumulated to $u_β$.

Complexity. The worst case compression cost in Algorithm 2.7 is $O(N^4)$, where $|\text{Near}(α)| = (N/m)$ for all $α$. The best case occurs when each $\text{Near}(α)$ only contains $α$ itself. We fix the rank $s$ and leaf size $m$. The tree has $O(N/m)$ leaf nodes and $O(N/m)$ interior nodes, so in the best case, overall $N2S$ has $O(2ms(N/m) + 2s^2(N/m))$ work, $S2S$ has $O(2s^2(N/m))$ work, $S2N$ has $O(2ms(N/m) + 2s^2(N/m))$ work, and $L2L$ has $O(2m^2(N/m))$. When $s$ and $m$ are held constant, the total work is $O(N)$ per right hand side. In GOFMM, this is controlled by the budget.

2.3 Shared memory parallelism

In H-matrix methods and FMM, the main algorithmic pattern is a tree traversal. A traversal may exhibit high parallelism at the leaf level, but the parallelism typically diminishes near the root level due to the dependencies. In addition, if the workload per tree node varies, load balancing becomes an issue. Most static scheduling codes employ level-by-level traversals, which introduces unnecessary synchronizations. In GOFMM, we observe significant workload variations during the compression (Algorithm 2.6) and during the evaluation (tasks N2S and S2N).

One solution is to exploit parallelism in finer granularity. For example, when the number of tree nodes in the single tree level is less than the number of cores, we can use multi-threaded BLAS/LAPACK on a single tree node. However, this is insufficient if the workload does not increase significantly (e.g. growing with $|α|$) while approaching the root. (That is, the workload must be within the strong scaling range of BLAS/LAPACK to be efficient).

To partially address these challenges, we abandon the convenient level-by-level traversal and explore an out-of-order approach using dynamic scheduling. To this end, we test two approaches and compare them with a level-by-level traversal. In the first approach, we
introduce a self-contained runtime system. In the second approach we test the same ideas with OpenMP’s omp task depend feature.

**Dependency analysis.** Recursive preorder and postorder traversals inherently encode Read/Write dependencies between tree nodes. Following Algorithm 2.2 and Algorithm 2.7, we can describe dependencies between different tasks. However, due to dynamic granularity of tasks we need a data flow analysis at runtime. For example, dependencies between N2S and S2S cannot be discovered at compile time, because the RAW (read after write) dependencies on \( \tilde{w}_\alpha \) are computed by neighbors \( N(\alpha) \). In order to build dependencies at runtime as a direct acyclic graph (DAG), we perform a symbolic execution on Algorithm 2.2 and Algorithm 2.7. For simplicity, below we just discuss the evaluation phase for the HSS case (the FMM case is more involved).

Figure 3 depicts task dependencies (by tasks we mean algorithmic tasks defined in Table 2) during the evaluation phase Algorithm 2.7 for N2S, S2S and S2N where the off-diagonals are low-rank (HSS) with \( S = 0 \). This task dependency graph is generated by our runtime using symbolic traversals. The N2S, S2S, and S2N execution order is performed on a binary tree6.

We use three symbolic tree traversals in Algorithm 2.7. In the first traversal (postorder) we find that \( \tilde{w}_1 \) is written by \( \alpha \). Going from \( \tilde{w}_1 \) to \( \tilde{w}_\beta \), we annotate that \( \tilde{w}_1 \) is read by \( \beta \), i.e., \( \tilde{w}_\beta = P_{\beta\{[\tilde{w}_1;\tilde{w}_1]\}} \). This RAW dependency is an edge from 1 to \( \beta \) in the DAG.

Inter-task dependencies are discovered by the symbolic execution of the yellow tree. At node \( \beta \) (in yellow), the relation \( \tilde{w}_\alpha = K_{\beta\alpha} \tilde{w}_\beta \) will read \( \tilde{w}_\beta \). Again this is a RAW dependency, hence the edge from the blue to the yellow \( \alpha \). The whole dependency graph for steps 1–3 is built after the green postorder traversal. Step 4 in Algorithm 2.7 is independent of steps 1–3. Although this runtime data flow analysis has some overhead, the amount is almost negligible (< 1%) compared to the total execution time.

6Execution order from left to right: dependencies are easier to follow if one rotates the page by 90° counter-clockwise.

**Runtime system.** With a dependency graph, scheduling can be done in static or dynamic fashion. Due to unknown adaptive rank \( s \) at compile time, we implement a light-weight dynamic Heterogeneous Earliest Finish Time (HEFT) [41] using OpenMP threads. Each worker (thread) in the runtime system can use more than one physical core with either a nested OpenMP construct or by employing a device (accelerator) as a slave. Tasks that satisfy all dependencies in the dependency graph will be dispatched to a “ready” queue. Each worker keeps consuming tasks in its own ready queue until no tasks are left.

Although we can estimate a cost for each task3 in Table 2, the execution time of a task on a normal worker (or one with an accelerator) depends on the problem and can only be determined at runtime. The HEFT schedule is implemented using an estimated finish time of all pending tasks in a specific worker’s ready queue. Each task dispatched from the dependency graph is assigned to a ready queue such that the maximum estimated finish time of each queue is minimized. For the case where the estimation is inaccurate, we also implement a job stealing mechanism.

**Other parallel implementation.** We briefly introduce other possible parallel implementations and conduct a strong scaling experiment in §4. Here we implemented parallel level-by-level traversals for all tasks that require preorder and postorder traversals and do not exploit out-of-order parallelism. For tasks that can be executed in any order, we simply use omp parallel for with dynamic scheduling. If there are not enough tree nodes in a tree level, we use nested parallelism with inner OpenMP constructs and multi-threaded BLAS/LAPACK.

The omp task version is implemented using recursive preorder or postorder traversals. Due to the overhead of the deep call stack, this implementation can be much slower than others. Although we tested it, we do not report results because it is not competitive.

We also implemented (and report results for) omp task depend, since OpenMP-4.5 supports task parallelism with dependencies. However there are two issues. First, omp task depend requires all dependencies to be known at compile time, which is not the case for the FMM (tasks N2S and S2S). Second, without knowledge of the estimated finish time, the OpenMP scheduler will be suboptimal. Finally for CPU-GPU hybrid architectures, scheduling GPU tasks purely with omp task can be very challenging.

**CPU-GPU hybrid.** GPUs usually offer high computing capacity, but performance can easily be bounded by the PCI-E bandwidth. Because most computations in Algorithm 2.2 are complex and memory bound4, we do not use GPUs for the compression. Instead we only pre-fetch submatrices \( K_{\beta\alpha} \) and \( K_{\alpha\beta} \) to the device memory to overlap with computations on the host (CPUs). During the evaluation, our runtime will decide—depending on the number of FLOPS—whether to issue a batch of tasks (up to 8) to the GPU in concurrent (using stream). This usually occurs in N2S and S2N where the size of cublasXgemm is bounded by \( s \) and \( m \). Furthermore, to hide communication time between CPU and GPU, all arguments of the next task in queue are pre-fetched using asynchronous communication for pipelining. Finally, because a worker with a GPU is usually 50X

3We divide costs for tasks by the theoretical peak FLOPS of the target architecture and a discount factor. For memory-bound tasks we use the theoretical MOPS instead.

4Although GEO5 and TRS8 can be performed on GPUs with MAGMA (http://icl.cs.utk.edu/magma/) and cublas, we find this inefficient for our methods.
to 100x more capable than others, we disable job stealing balancing for GPU workers. This optimization prevents the GPU from idling.

Distributed parallelism. In this work, we do not discuss how to parallelize GOFMM in a distributed environment. The MPI extension requires new algorithms, which will be discussed in the future work of GOFMM. The basic philosophy of MPI parallelism follows [11, 32, 48], which include distributed tree traversal, distributed nearest-neighbor search, local essential trees for reducing communication, and distributed linear algebra operations. New challenges include parallelizing matrix access, integrating the task-scheduling with MPI, accounting for off-diagonal dependencies from other ranks, and load-balancing. Inter-process job stealing may also result in extra communication.

3 EXPERIMENTAL SETUP
We perform experiments on Haswell, KNL, ARM, and NVIDIA GPU architectures with four different setups to examine the accuracy and efficiency of our methods. We demonstrate (1) the robustness and effectiveness of our geometry-oblivious FMM, (2) the scalability of our runtime system against other parallel schemes, (3) the accuracy and cost comparison with other software, and (4) the absolute efficiency (in percentage of peak performance).

Implementation and hardware. Please refer to §5.2 for all configuration in the reproducibility artifact section. GOFMM is implemented in C++ and CUDA, employing OpenMP for shared memory parallelism. The source code of GOFMM can be found in the Github repository (https://github.com/ChenhanYu/hmlp). Our tests were conducted on TACC’s Lonestar 5, (two 12-core, 2.6GHz, Xeon E5-2690 v3 ”Haswell”), TACC’s Stampede 2 (68-core, 1.4GHz, Xeon Phi 7250 ”KNL”) and CSCS’s Piz Daint (12-core, 2.3GHz, Xeon E5-2650 v3 and NVIDIA Tesla P100).

Matrices. We generated 22 matrices emulating different problems. K02 is a 2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. The Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid. K03 has the same setup with the oscillatory Helmholtz operator and 10 points per wave length. K04-K10 are kernel matrices in six dimensions (Gaussians with different bandwidths, narrow and wide; Laplacian Green’s function, polynomial and cosine-similarity). K12-K14 are 2D advection-diffusion operators on a regular grid with highly variable coefficients. K15,K16 are 2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients. K17 is a 3D pseudo-spectral operator with variable coefficients. K18 is the inverse squared Laplacian in 3D with variable coefficients. G01–G05 are the inverse Laplacian of the powersim, poli_large, rgg_n_2_16_s0.denormal, and conf6_0-8x8-30 graphs from UFL (http://yifanhu.net/GALLERY/GRAPHS/search.html).

K02–K03, K12–K14, and K18 resemble inverse covariance matrices and Hessian operators from optimization and uncertainty quantification problems. K04–K10 resemble classical kernel/Green function matrices but in high dimensions. K15–K17 resemble pseudo-spectral operators. G01–G05 (N = 15838, 15575, 65536, 89400, 49152) are graphs for which we do not have geometric information. For K02–K18, we use N = 65536 if not specified.

Also, we use kernel matrices from machine learning: COVTYPE (100K, 54D, cartographic variables); and Higgs (500K, 28D, physics) [28]; MNIST (60K, 780D, digit recognition) [9]. For these datasets, we use a Gaussian kernel with bandwidth h. GOFMM supports both double and single precision. All experiments with matrices K02–K18 and G01–G05 are in single precision. The results for COVTYPE, Higgs, MNIST are in double precision. In the Github repository, we provide a MATLAB script to generate K02–K18. For real world datasets and graphs, we provide the link to their original sources.

Parameter selection and accuracy metrics. We control m (leaf node size), s (maximum rank), τ (adaptive tolerance), κ (number of neighbors), budget (a key parameter for amount of direct evaluations and for switching between HSS and FMM) and partitioning (Kernel, Angle, Lexicographic, geometric, random). We use m =256–512; on average this gives good overall time. The adaptive tolerance τ reflects the error of the subsampled block and may not correspond to the output error ε2. Depending on the problem, τ may underestimate the rank. Similarly, this may occur in HODLR, STRUMPACK and ASKIT. We use τ between 1E-2 and 1E-7, s = m, k = 32 and 3% budget. To enforce a HSS approximation, we use 0% budget. The Gaussian bandwidth values are taken from [31] and produce optimal learning rates.

Throughout we use relative error ε2 defined as the following

$$
\varepsilon_2 = \frac{\|\tilde{K}w - Kw\|_F}{\|Kw\|_F}, \quad \text{where } w \in \mathbb{R}^{N \times r} \tag{11}
$$

This metric requires $O(rN^2)$ work; to reduce the computational effort we instead sample 100 rows of K. In all tables, we use “Comp” and “Eval” to refer the the compression and evaluation time in seconds, and “GFs” to GFLOPS per node.

4 EMPIRICAL RESULTS
We label all experiments from #1 to #46 in tables and figures. We perform strong scaling results on a single Haswell and KNL node in Figure 4, comparing different scheduling schemes. In Figure 5, we examine the accuracy of GOFMM for the different matrices; notice that not all 22 matrices admit good hierarchical low-rank structures in the original order (lexicographic). In Figure 6, we compare FMM (S ≠ 0 in (1)) to HSS (S = 0) and show an example in which increasing direct evaluations in FMM results in higher accuracy and shorter wall-clock time. In Figure 7, we present a comparison between five permutation schemes; matrix-defined Gram distances work quite well.

For reference, we compare GOFMM to three other codes: HODLR and STRUMPACK (S = 0 in these codes) in Table 3 and ASKIT (high-d FMM) in Table 4. The two first codes do not permute K. ASKIT is similar to GOFMM but uses level-by-level traversals, does not produce a symmetric $\tilde{K}$, and requires points. Finally, we test GOFMM on four different architectures in Table 5; the performance of GOFMM correlates with the performance of BLAS/LAPACK.

Strong scaling (Figure 4). In #1, #2, #3, and #4, we use a 24-core Haswell and a 68-core KNL to perform strong scaling experiments. Each set of experiments contains 6 bars including 3 different parallel schemes on both Algorithm 2.2 and Algorithm 2.7. The blue dot indicates the absolute efficiency (ratio to the peak) of our evaluation using dynamic scheduling. #1 and #2 require 12% budget with average rank 487 to achieve 2E–3. This compute-bound problem can

Chenhun D. Yu, James Levitt, Severin Reiz, and George Biros
reach 65% peak performance on Haswell and 33% on KNL. However, #3 and #4 only require 3% budget with average rank 35 to achieve SE–5. As a result, this memory-bound problem does not scale (46% and 85%) very well. In #4, we can even observe slow down from 34-core to 68-core. This is because the wall-clock time is bounded by the task in the critical path; thus, increasing the number of cores does not help.

Throughout, we can observe that the wall-clock time for compression is less than the level-by-level and omp task traversals. While the work of SKEL is bounded by 2s, parallel GEQFP3 in the level-by-level traversal does not scale (especially on KNL). On the other hand, task based implementations can execute COEF and Kba out-of-order to maintain the parallelism. Our wall-clock time is better than omp task since we use the cost-estimate model for scheduling.

Accuracy (Figure 5). We conduct #5 to examine the accuracy of GOFMM (up to single precision). Given m512, s512 and r512, we report relative error $e_2$ on K02–18 and G01–G05 using the Angle distance with two tolerances: 1E–2 in blue and 1E–5 in green. Throughout, except for K06, K15–K17 (high rank), K13, K14 (underestimating the rank), and G01–G03 (requiring smaller leaf size $m$), other matrices can usually achieve high accuracy with tolerance 1E–5 (0.9% in compression and 0.2s in evaluation). Our adaptive ID underestimates the rank of K13 and K14 such that $e_2$ is high. By imposing a smaller tolerance 1E–10 (yellow plots), both matrices reach 1E–5 (1s in compression and 0.2s in evaluation). K6, K15–K17 have high ranks in the off-diagonal blocks; thus they cannot be compressed with s512 and 3% budget. G01–G03 requires direct evaluation in the off-diagonal blocks to reach high accuracy. When we reduce the leaf node size from 512 to 64, we can still reach 1E–5 (orange plots). However, decreasing leaf size to 64 results in a longer wall-clock time (0.8s in evaluation), because small $m$ hurts performance. Overall, we can observe that GOFMM can quite robustly discover low-rank plus sparse structure from different SPD matrices. We now investigate how increasing the cost (either with higher rank or more direct evaluations) can improve accuracy.

Comparison between FMM and HSS (Figure 6). We use #6, #7, and #8 to show that even with more evaluations, FMM can be faster than HSS for the same accuracy. For HSS the relative error in #6 (blue plots) plateaus at SE–4. Further increasing rank from 256 to 512 (or even 1,024) results in $O(s^3)$ work (green bars). Using a combination of low-rank ($s^4$) and 3% direct evaluation, FMM can achieve higher accuracy with little increment in the evaluation time (compression time remains the same). Similarly, in #8 we can observe that by using s512 and 3% budget we achieve better accuracy than the HSS approximation (s2048) in less time.

---

1The average rank of #4 is too small. Except for 12L tasks, other tasks can only reach about 5% of the peak during the evaluation. We suspect that MKL’s GEQFP uses a 30 x 16 micro kernel to perform a 30 x 256 x 16 rank-k update each time. For an $m \times k \times n$ GOFMM to be efficient, $m$ and $n$ usually need to be at least four times of the micro-kernel size in each way. In #4, many GOFMMs have $m < 30$. Still the micro-kernel must compute $2 \times 30 \times 256 \times 16$ FLOPS. These sparse FLOPS are not counted in our experiments.
Permutations (Figure 7). Here we test different permutations (#9, #10, #11, and #12) to discuss the different distances in GOFMM. In each set of experiments, we present relative error (blue plots) and average rank (green bars) for five different schemes. The first two schemes use lexicographic or random order to recursively permute \( K \). Since there is no distance defined, these two schemes can only use HSS approximation. The Angle and Kernel distance use the corresponding Gram distances §2.1. Finally, we also use standard geometric distance from points. For the last three schemes, we use \( \kappa = 32 \) and 3% budget. Overall, we can observe that the distance metric is important in discovering low-rank structure and improving accuracy. For example, in #9, Kernel and Geometric show much lower average rank than others. In #10 and #11, although the average ranks are not significantly different, distance-based methods usually have higher accuracy. Finally, we observe for matrix GOFMM in #12 where no coordinate information exists, our geometry-oblivious methods can still compress the matrix. The lexicographic permutation has very low rank, the error is large. This is because the uniform samples for the low-rank approximation are poor. Angle and Kernel distance use neighbors for importance sampling, which greatly improves the quality of the low-rank approximation.

Comparison to existing software (Table 3, Table 4). We compare our methods to HODLR [3], STRUMPACK [38], and ASKIT [33]. Let us summarize some key differences. HODLR uses the Adaptive Cross Approximation (ACA, partial pivoted LU) for constructing the low-rank blocks (using the Eigen library). Its evaluation requires \( O(N \log N) \) work since the \( U, V \) matrices are not nested.

| Parameters | ASKIT | GOFMM |
|------------|-------|-------|
| # case     | \( N \) | \( \tau \) | Comp | Eval | Comp | Eval |
| 19         | K04   | 36 864 | 1E–3 | 2E–4 | 0.3  | 2E–2 |
| 20         | K04   | 36 864 | 1E–6 | 8E–7 | 1.4  | 4E–2 |
| 21         | K04   | 65 536 | 1E–3 | 2E–4 | 1.0  | 4E–2 |
| 22         | K04   | 65 536 | 1E–6 | 7E–7 | 2.2  | 8E–2 |
| 23         | K06   | 36 864 | 1E–3 | 4E–2 | 6.6  | 4E–2 |
| 24         | K06   | 36 864 | 1E–6 | 2E–2 | 7.4  | 6E–2 |
| 25         | K06   | 65 536 | 1E–3 | 4E–2 | 11.1 | 1E–1 |
| 26         | K06   | 65 536 | 1E–6 | 5E–2 | 12.0 | 1E–1 |

Figure 7 Accuracy (left y-axis) and rank (right, x-axis) comparison: Lexicographic, Random, Kernel 2-norm, Angle and Geometric. We use \( \tau = 1E–7 \), \( s = 512 \), and \( m = 64 \). For methods that define distance, we use \( k = 32 \) and 3% budget. GOFMM is a graph Laplacian; thus, using Geometric distance is impossible.

Table 4 Wall-clock time (in seconds) and accuracy \( e_2 \) comparison with ASKIT.

For both methods, we use \( \kappa = 32 \), \( m = s = 512 \), and \( r = 1 \). ASKIT uses the \( \tau \) reported in the table, and we adjust the tolerance of GOFMM to match the accuracy. For all experiments, GOFMM uses 7% budget. The amount of direct evaluation performed by ASKIT is decided by \( \kappa \).

STRUMPACK constructs an HSS representation in \( O(N \log N) \) work. This is done by using a randomized ID according to [27]. We used their black-box compression routine with a uniform random distribution and a Householder rank-revealing QR. Once the matrix is compressed, the evaluation time is \( O(N) \) per right hand side. STRUMPACK supports multiple right hand sides. ASKIT’s FMM evaluation has similar complexity as GOFMM, but the amount of direct evaluation is only decided by \( \kappa \). For GOFMM, we further introduce the budget to restrict the cost. For all comparisons, we try to match the accuracy by controlling different parameters (\( \tau \) and \( s \)). Notice that ASKIT and STRUMPACK support MPI, whereas GOFMM does not. We have not used MPI for distributed environment in our experiments.

In Table 3, we target final accuracy \( e_2 = 1E–4 \). GOFMM uses Angle distance for neighbor search and tree partitioning. HODLR and STRUMPACK do not have built-in partitioning schemes for dense matrices. STRUMPACK fails to compress K04 (Gaussian kernel in 6D) and K07 (Laplace kernel in 6D). This is because the lexicographic order does not admit a good \( H \)-matrix approximation. The matrix needs to be permuted. K17 is difficult to compress with a pure hierarchical low-rank matrix. Finally, GOFMM performs better when \( S > 0 \). HODLR and STRUMPACK must increase the off-diagonal ranks to match the accuracy and thus the cost increases. With a sparse correction S, GOFMM is about 25× faster in compression and about 1.5× faster in evaluation.

In Table 4, we compare GOFMM (with geometric distances) to ASKIT. ASKIT uses level-by-level traversals in both compression and evaluation. Since ASKIT only evaluates a single right hand side, we use \( r = 1 \). The compression time is inconclusive for #19–#22; the average ranks used in two methods are quite different. The benefit of out-of-order traversal appears in #23–#26 where both methods reach the maximum rank \( s \). The speedup in evaluation is not significant, but GOFMM can get up to 2× speedup in compression.

Different architectures. In Table 5, we present wall-clock time and GFLOPS of GOFMM on four architectures for different problems. We want to show that the efficiency of GOFMM is portable and only relies on BLAS/LAPACK libraries.

In #27 and #28, we show that a quad-core ARM processor can handle up to 100K fast matrix-multiplication. Because we only have limited memory (2GB) and storage (8GB), in GOFMM we compute \( K_{ij} \) on the fly (in detail, we compute \( K_{ij} \) using the...
2-norm expansion). #27 takes much longer than #28 because the cost of evaluating $K_{ij}$ is proportional to the point dimensions of the dataset (MNIST in 780D and COVTYPE in 54D). Because there is no active cooling on the board, the ARM processor gets overheated and is forced to reduce its clockrate. That is why we can only reach 3% on the CPU (see §2.3).

Table 5 Accuracy $\varepsilon_2$, wall-clock time (in seconds) and efficiency (in GFLOPS) on four architectures. Because our ARM platform only has a 8GB SD card and 2GB DRAM, we only perform kernel matrices ($K_{ij}$ computed on the fly) with small $r$ and $s$. Note that in the CPU+GPU experiment, the compression is run on the CPU (see §2.3).

| #  | Arch | Budget | $\varepsilon_2$ | Comp | GFs Eval | GFs |
|----|------|--------|---------------|------|---------|------|
| 27 | ARM  | 5%     | 5E-3          | 285  | 3       | 520  |
| 28 | COVTYPE100K | 1% | k32, m512, s128, r256 | 5%   | 8E-4   | 71   | 2     | 61   |
| 29 | CPU  | 12%    | 2E-3          | 30   | 3       | 4.1  | 679  |
| 30 | CPU+GPU | 12% | 3E-3          | 33   | 29      | 1.7  | 1952 |
| 31 | KNL  | 12%    | 2E-3          | 48   | 25      | 3.2  | 1125 |
| 32 | K02, N65536, k32, m512, s128, r512 | 0.3% | 2E-1          | 102  | 18      | 3.3  | 592  |
| 33 | CPU  | 0.3%   | 2E-1          | 180  | 12      | 1.7  | 1147 |
| 34 | KNL  | 0.3%   | 2E-1          | 121  | 17      | 2.2  | 872  |
| 35 | K03, N65536, k32, m512, s128, r512 | 3%  | 9E-5          | 1    | 25      | 0.2  | 889  |
| 36 | CPU+GPU | 3%   | 1E-4          | 2    | 12      | 0.1  | 2175 |
| 37 | KNL  | 3%     | 1E-4          | 3    | 11      | 0.3  | 530  |
| 38 | K13, N65536, k32, m512, s128, r512 | 10%  | 2E-1          | 6.0  | 81      | 1.1  | 1495 |
| 39 | CPU+GPU | 10%  | 2E-1          | 7.8  | 62      | 0.66 | 2514 |
| 40 | KNL  | 10%    | 2E-1          | 9.2  | 53      | 1.3  | 1549 |
| 41 | K04, N89400, k32, m512, s512, r512 | 3%  | 4E-5          | 4.8  | 37      | 0.5  | 1122 |
| 42 | CPU+GPU | 3%  | 3E-5          | 7.9  | 19      | 0.53 | 962  |
| 43 | KNL  | 3%     | 5E-5          | 11.8 | 9.1     | 0.6  | 741  |
| 44 | G04, N89400, k32, m512, s512, r512 | 3%  | 4E-6          | 1.8  | 21      | 0.3  | 787  |
| 45 | CPU+GPU | 3%  | 4E-6          | 4.0  | 10      | 0.13 | 2277 |
| 46 | KNL  | 3%     | 4E-6          | 4.2  | 9       | 1.5  | 215  |

5 CONCLUSIONS

By using the Gramian vector space for SPD matrices, we defined distances between rows and columns of $K$ using only matrix values. Using the distances, we introduced GFOMM and $\mathcal{H}$-matrix scheme that can be used to compress arbitrary SPD matrices (but without accuracy guarantees). These algorithms are applied black-box for various problems in computational science and we observe that the approach can be very attractive. In GFOMM we use a shared-memory runtime system that performs out-of-order scheduling in parallel to resolve the dynamic workload due to adaptive ranks and the parallelism-diminishing issue during tree traversals. Our future work will focus on the distributed algorithms and the hierarchical matrix factorization based on our method. We also plan to improve the sampling and pruning quality and to reduce the number of parameters that users need to provide.

ACKNOWLEDGMENT

This material is based upon work supported by AFSOR grants FA9550-17-1-0190; by NSF grant CCF-1337939; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0015018 and DE-SC0009286; by NIH grant 10042242; by DARPA grant W911NF-115-2-0121; and by the Technische Universität München—Institute for Advanced Study, funded by the German Excellence Initiative (and the European Union Seventh Framework Program under grant agreement 291763), as well as the German Academic Exchange Service (DAAD). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFSOR, DOE, NIH, DARPA, and NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.
APPENDIX

5.1 Abstract
This artifact description comprises the source code, datasets, and installation instruction on a GitHub repository that can be used to reproduce results for this SC’17 paper. We also provide all hardware and software configuration in §5.2.

5.2 Description
Checklist. We briefly describe all meta information. The GOFMM software implements an algebraic FMM (Fast Multipole Method) with geometry-oblivious techniques that generalize to generic SPD matrices. The only requirement is the capability to sample (access the value of) any matrix element.

- **Program.** GOFMM is developed in C++ (with C++11 features) and CUDA (Compute Unified Device Architecture from NVIDIA), employing OpenMP for shared memory parallelism using a self-contained runtime system.

- **Hardware.** We conducted experiments on Lonestar5 (two 12-core, 2.6GHz, Xeon E5-2650 v3 “Haswell” per node) and Stampede (68-core, 1.4GHz, Xeon Phi 7250 “KNL” per node) clusters at the Texas Advanced Computing Center, Piz Daint (12-core, 2.3GHz, Xeon E5-2650 v3 and NVIDIA Tesla P100) at the Swiss National Supercomputing Centre, and an Intrinsyc Open-Q 820 Development Kit (quad-core, 2.2GHz Qualcomm Kyro).

- **Compilation.** All software (including Hodg, STRUMPACK, and ASKLT) are compiled with intel-16.0 -03 on Lonestar5 and Piz Daint. Stampede uses intel-17.0 -03 -xMIC-AVX512. The GPU part uses nvcc-8.0 -03 -arch=sm_60. For Open-Q 820, we cross compile our software with Android NDK\(^6\) using gcc-4.9 -03. All CPU and KNL BLAS/LAPACK routines use Intel MKL. GPU BLAS routines use CUBLAS; on ARM we use QSML (Qualcomm Snapdragon Math Library)\(^7\). KNL experiments use Cache-Quadrant configuration. OpenMP uses OMP_PROC_BIND=spread.

- **Datasets.** Our 22 matrices K02–G05 can be generated using MATLAB scripts (/artifact/sc17gofmm/datasets/*.m, provided in the repository). For the five graphs G01–G05 and other real world datasets we use for generating kernel matrices, we provide URLs in §3.

- **Output.** The program reports execution time, total GFLOPS of the compression and evaluation phases, and accuracy $\epsilon_2$ of the first 10 entries and averaged over 100 entries.

- **Experiment workflow.** git clone projects; generate datasets using the matlab script; run test scripts; observe the results.

How delivered. GOFMM can be found in the GitHub repository (https://github.com/ChenhanYu/hmlp) as a set of high level routines and data structures in project HMLP (High-Performance Machine Learning Primitives). The software comprises code, build, and evaluation instructions, and is provided under GPL-3.0 license.

Hardware dependencies. For adequate reproducibility, we suggest that reproducers use the same environment as mentioned above in the Hardware and Compilation sections. Notice that we report absolute GFLOPS (Giga Floating Points Operations per Second) and the ratio to peak performance in the paper. To approximately reproduce the same results on a different environment, the reproducer should seek a platform that has similar capabilities. The theoretical peak performance\(^8\) in double precision is 998 GFLOPS per Haswell node, 3046 GFLOPS per KNL node, (4700 + 416) GFLOPS per Tesla P100 node, and 35.2 GFLOPS per Open-Q820. The peak GFLOPS doubles for single precision computations.

Software dependencies. Compilation of GOFMM requires generic C/C++ compilers that support c++11 features (-std=c++11) and OpenMP (-fopenmp for GNU or -openmp for Intel compilers). GOFMM also requires full functionality of BLAS and LAPACK routines.

5.3 Installation
Given the repository URL, you should be able to clone the release branch of the repository. The first step is to source set_env.sh or source set_env_tacc.sh (for most TACC machines) to set up compilers, BLAS/LAPACK library, and architecture. If no error occurs, use the following instructions for compilation.

```
source set_env.sh
mkdir build
cc build
cmake ..
make install
```

Linux and Unix. If environment variables CC, CXX, MKLROOT (or OPENBLASROOT) are all set up properly, then no further modification is required for set_env.sh. Otherwise, you must export the proper values for all variables in the required region of set_env.sh first. GOFMM detects the compiler vendor by environment variables CC and CXX and uses cmake for compilation. In set_env.sh, modify the following two lines to set up the desired C/C++ compilers.

```
export CC = $(CC) % icc or gcc
export CXX = $(CXX) % icpc or g++
```

Mac OSX. If you are using Mac OSX, you must first install a generic GNU or Intel C/C++ compiler. CC in Mac OSX is a soft link to the default clang compiler, which is currently not supported by GOFMM. To use Intel compilers, you must modify the following line in set_env.sh.

```
export HMP_USE_INTEL = true % Intel (true) or GNU (false)
```

GOFMM will try to use FindBLAS in cmake to locate and link to a proper BLAS/LAPACK library. However, it is better to manually provide the paths to the root directories of Intel MKL or OpenBLAS.

```
export $(MKLROOT) = $(MKLROOT) % Intel MKL
export $(OPENBLASROOT) = $(OPENBLASROOT) % OpenBLAS
```

---

\(^6\)https://developer.android.com/ndk
\(^7\)https://developer.qualcomm.com/software/snapdragon-math-libraries
\(^8\)We estimate the peak according to the clockrate and the FMA throughput. For 24 Haswell cores, 998 = 2 x 12 x 2.6 x 16. For 68 KNL cores, 3046 = 68 x 1.34 x 32. For 4 ARM cores, 35.2 = 4 x 2.2 x 4. The peak of P100 is reported as 4.7 TFLOPS. For reference, MKL GEMM can achieve 87% on a Haswell node and 60% on a KNL node. QSML GEMM can achieve 89% on Open-Q 820. cublasgemm can achieve 95% on P100. We assume two KNL VPUs can dual issue 99Mflop\(^7\). However, Intel processors may have a different frequency while fully issuing FMA, and the clockrate may drop to 1.0 GHz. This may be the reason why MKL GEMM can only achieve 2.1 TFLOPS on KNL.
GOFMM employs a self-contained runtime system to perform dynamic scheduling. The number of workers is controlled by the environment variable OMP_NUM_THREADS. Depending on the architecture, we suggest that users set this value to be the same as the number of physical cores.

**Advanced options.** In the advanced options of set_env.sh, there are three options for the host architecture (ARM, x86-64, or KNL). Users must choose at least one major and minor architecture to compile. This can be arm/armv8a, x86-64/haswell or mic/knl.

### Compilation with Intel Knights Landing (KNL)

For KNL, compilation must use Intel compilers (i.e., CC = icc and CXX = icpc). Export OMP_NUM_THREADS = 68. Select the mic/knl combination in the advanced options.

### Compilation with NVIDIA GPUs

In order to compile the CUDA code for the hybrid CPU-GPU implementation, the following variables have to be exported in the advanced options of set_env.sh.

```bash
export HMLP_ARCH_MAJOR=x86_64 % default major arch
export HMLP_ARCH_MINOR=haswell % default using AVX2
export HMLP_ARCH_MAJOR=mic % Intel Xeon Phi
export HMLP_ARCH_MINOR=knl % Knights Landing AVX512
export HMLP_ARCH_MAJOR=arm % native/cross compilation
export HMLP_ARCH_MINOR=armv8a %
```

The user must also set up all variables in the required section appropriately for the host machine.

**Native/cross compilation for ARM.** If your ARM device runs with an OS that has native compilers and cmake support, then the installation instructions above should work just fine. However, while your target runs an Android OS, which currently does not have a native C/C++ compiler, you will need to cross compile this software on your Linux or OSX first. Although there are many ways to do cross compilation, we suggest that users follow these instructions:

- Install Android Studio with LLDB, cmake, and NDK support.
- Create a stand-alone toolchain from NDK.
- Install adb (Android Debug Bridge).
- Download and export the proper path of QSMIL (Qualcomm Snapdragon Math Library) or OpenBLAS.
- Use arm/armv8a in the advanced options of set_env.sh.
- Compile with cmake. It will look for your ARM gcc/g++, ar and ranlib support.
- Use the following instructions to push the executable and scripts in /build/bin to the Android ARM device.

```bash
adb devices
adb push /build/bin/* /data/local/tmp
adb shell
cd /data/local/tmp
```

Executing the default script will invoke the GOFMM executable (artifact_sc17gofmm.x) on a small 5K-by-5K matrix. To execute GOFMM with custom settings, change the matrixtype variable in the script to dense or kernel and edit other options in the bash script appropriately. To reproduce other experiments in this paper, you must first generate binary datasets using the MATLAB scripts we provide in /datasets. See the following instructions for generating datasets with different problem sizes.

### 5.4 Datasets

The 22 matrices we use can be generated using MATLAB scripts with the corresponding coordinates or graphs. To generate and store the first 17 matrices (K02–K18) as binary files in single precision, execute the matlab script spdmatrices.m in the artifact directory.

```bash
cd hmlp/build/bin/datasets
cd hmlpObuildObinOdatasets
matlab -nodisplay -nodesktop -r "spdmatrices"
```

The variable n in spdmatrices.m controls the matrix size N. Notice that depending on the problem, the matrix size N may scale with \(O(n^2)\) or \(O(n^3)\). The last five graphs (G01–G05) we use in the paper are reported in §5 with URLs. Users can compute and store their inverse graph Laplacian to reproduce the results.

### 5.5 Experiment workflow

With the repository URL, git clone projects. Generate datasets using the provided MATLAB script. Compile GOFMM with the instructions in §5.3. Run the test script for each architecture. Observe the results.

### 5.6 Evaluation and expected results

For x86-64, ARM, and KNL execution, the program will start from the iterative ANN. The accuracy is reported in every iteration. Once the neighbor search is done (or skipped), the metric ball tree partitioning follows. The program reports execution time and total FLOPS of the compression and evaluation phases. Finally, the accuracy \(\varepsilon_2\) is reported in two parts: the error of the first 10 entries, and the average error of 100 entries. Notice that in a CPU-GPU hybrid environment, GOFMM will first try to detect the available GPU device. If successful, the device name and the available global memory size should be displayed. The rest of the execution is the same as for other architectures.