Black Box Hierarchical Approximations for SPD Matrices

Author: Severin Maximilian Reiz
1st examiner: Prof. George Biros
2nd examiner: Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor(s): James Levitt
Thesis handed in on: April 14, 2017
I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

April 18, 2017

Severin Maximilian Reiz
Acknowledgments

This work was carried out as part of a visiting research stay in the Parallel Algorithms for Data Analysis and Simulation Lab (PADAS) in the Institute of Computational Engineering and Sciences (ICES) at the University of Texas at Austin. The work was performed between October 2016 and April 2017 and was kindly supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD). I would like to express my gratitude to George Biros for welcoming me and introducing me to the topic with ideas and countless explanations on current (and reliable) solutions. George was leading progress with (almost) weekly meetings where he was open to new ideas as well as coming up with new features and goals. Furthermore, I want to thank H.J. Bungartz to support the project. For me the stay was a great experience and helped me develop both professionally and personally.

Furthermore, I would like to thank James Levitt who guided me in steps through algorithms, answered all my many questions and was always present. His ideas and comments were very helpful to me. Furthermore, I would like to thank Chenhan D. Yu for providing a good software suite for the parallel framework, his particular expertise on the current software ASKIT and the many items he solved for GOFMM, a software package developed along this thesis.
Abstract

Kernels are a widely used tool in a wide range of learning methods, including density estimation, dimension reduction and supervised learning tasks. A key computational bottleneck in training and evaluating these methods is the kernel matrix – the matrix of all pairwise kernel evaluations between points. Members of the research group have previously published ASKIT: Approximate kernel independent treecode, an efficient approximation method for evaluating matrix-vector products with the kernel matrix. However, it requires as input data-points and a kernel function to generate entries required for the method. The task of this thesis is to generalize the method so that it can work black-box on symmetric positive definite (SPD) matrices.

We use research efforts from ASKIT – a high dimensional approach which uses nearest-neighbor pruning. Instead of a geometric knowledge of the point distribution, we employ the fact that every SPD matrix can be viewed as a Gramian matrix – where entries $K_{ij}$ can be built by inner products of an underlying set of Gram vectors \( \{ \phi_i \}_{i \in [1...N]} \), i.e. $K_{ij} = \langle \phi_i, \phi_j \rangle$. Using the Gram vector space we introduce two metrics for hierarchical partitioning and finding row neighbors.

We apply the approach to 22 matrices related to machine learning, stencil PDEs, spectral PDEs, inverse problems, and graph Laplacian operators. For almost all of them we observe good approximation behavior. A Python and C++ (task-based shared-memory parallel) framework called GOFMM is developed along this project.
Contents

Acknowledgements v
Abstract vii
1. Introduction 1

2. Theory 5
  2.1. Low-rank approximations ............................................. 5
    2.1.1. Geometric motivation from N-body problems ................. 5
    2.1.2. Algebraic compression algorithms ............................. 8
    2.1.3. Randomized linear algebra ................................... 10
  2.2. Hierarchical matrices .............................................. 11
    2.2.1. SPD matrix as gramian ....................................... 12
    2.2.2. Hierarchical matrix partitioning ............................ 12
    2.2.3. Hierarchical off diagonal low rank matrices ............. 14
  2.3. Arithmetic operations ............................................... 15
    2.3.1. Matrix vector product ...................................... 15
    2.3.2. FMM-like evaluations ...................................... 17
    2.3.3. Matrix inversion .......................................... 17
  2.4. Related work ....................................................... 17

3. Methods 19
  3.1. Geometry-oblivious techniques ................................... 19
    3.1.1. Matrix reordering ......................................... 19
    3.1.2. Neighbor search using randomized trees .................... 22
  3.2. Matrix compression ................................................. 23
    3.2.1. Compression algorithm .................................... 23
    3.2.2. Adaptive rank selection .................................. 24
    3.2.3. Target sampling .......................................... 24
  3.3. Evaluation .......................................................... 26
    3.3.1. Outgoing representation ................................... 26
    3.3.2. Pruning rule ............................................. 26
    3.3.3. Incoming representation .................................. 27
  3.4. Parallel schemes .................................................. 28
  3.5. Experimental setup ................................................ 30

4. Numerical Experiments 33
  4.1. Evaluation of Gram vector approach ............................. 33
    4.1.1. Hierarchical partitioning .................................. 33
## Contents

4.1.2. Sampling Strategy ................................................. 34  
4.2. Performance Results ............................................... 34  
  4.2.1. Evaluation Accuracy .......................................... 34  
  4.2.2. Comparison to other software ................................. 37  

5. Conclusion and future work 43  
  5.1. Conclusion .......................................................... 43  
  5.2. Future work ....................................................... 44  

Appendix 47  
A. Detailed Descriptions 47  
  A.1. Python code ...................................................... 47  
    A.1.1. Installation and Usage ..................................... 47  
    A.1.2. File structure .............................................. 47  
    A.1.3. Test Matrices ............................................... 47  
  A.2. C++ Framework ................................................... 48  
    A.2.1. Compilation ................................................ 48  
    A.2.2. Usage ........................................................ 49  
    A.2.3. Evaluation and expected result ............................ 50  

Bibliography 51
1. Introduction

This thesis describes a novel tunable algorithm for the approximation of dense symmetric positive definite (SPD) matrices. It can be used for compressing a dense matrix and accelerating matrix-vector multiplication operations (hereby referred to as \textit{matvec}). For example, a general \textit{matvec} can often be a bottleneck in applications due to a quadratic complexity in required computational work. In many physical problems however, we can find blocks -- typically off-diagonal terms -- that are compressible using structured (or low-rank or data-sparse) matrices.

Let $K \in \mathbb{R}^{N \times N}$ be a dense SPD matrix, i.e. $K = K^T$ 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 general \textit{matvec}. Using $O(N \log N)$ memory and work, we construct an approximation $\tilde{K}$ such that $\|K - \tilde{K}\| \leq \epsilon \|K\|$, where $\epsilon$ is a user-defined error tolerance. Assuming the evaluation of a single matrix entry $K_{ij}$ requires $O(1)$ work, a \textit{matvec} with $\tilde{K}$ requires $O(N \log N)$ or $O(N)$ work depending on the properties of $K$ and the variant of the compression. Our scheme belongs to the class of hierarchical matrix approximation methods.

Problem statement

Given any SPD matrix $K$, our task is to construct a hierarchically low-rank matrix $\tilde{K}$ such that $\|K - \tilde{K}\|/\|K\|$ is small. The only required input to our algorithm is a routine that returns $K_{IJ}$, for arbitrary row and column index sets $I$ and $J$. The constant in the complexity estimate depends on the structure of the underlying matrix. Let us remark and emphasize that our scheme cannot guarantee both accuracy and work complexity simultaneously since an arbitrary SPD matrix may not admit a good hierarchical matrix approximation (see subsection 4.2.2).

We say that a matrix $\tilde{K}$ has a hierarchical low-rank structure, i.e., $\tilde{K}$ is an $\mathcal{H}$-Matrix [36, 9], if

\[
\tilde{K} = D + S + UV, \quad (1.1)
\]

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 small dense matrices. An $\mathcal{H}$-Matrix \textit{matvec} requires $O(N \log N)$ work the constant depending 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
1. Introduction

tree data-structure) or by using graph partitioning techniques (if $K$ is sparse). In our novel algorithm we do not require such geometric information.

Background and significance

Dense SPD matrices appear in scientific computing, statistical inference, and data analytics. They appear in Cholesky and LU factorization [29], in Schur complement matrices for saddle point problems [11], in Hessian operators in optimization [58], in kernel methods for statistical learning [42, 31], and in $N$-body methods and integral equations [34, 36].

In many applications, the entries of the input matrix $K$ are given by $K_{ij} = K(x_i, x_j) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, where $K$ 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 by using expansions 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 $H$-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 [13, 44]).

Contributions

The algorithm 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 [42]. 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 this algorithm (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 different architectures. Finally, we compare with three state-of-the-art codes: HODLR, STRUMPACK, and ASKIT.

Our novel algorithm 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 described in [52, 50]. GOFMM currently supports three different measures of distance: geometric point-based (if available), Gram-space $\ell^2$ distance and Gram-space angle distance. Our algorithm GOFMM has support for matvecs with multiple vectors, which is useful for Monte-Carlo sampling, optimization, and block 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 thesis. Our method requires the ability to evaluate kernel 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 (1.1), this is equivalent to setting $D$ and $S$ to zero and constructing only $U$ and $V$. Examples include the CUR [47] factorization, the Nystrom approximation [64], the adaptive cross approximation [10], and randomized rank-revealing factorizations [54, 39]. 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 [56, 66], and the black-box FMM [24]. 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 $O(N^3)$ [1, 43].
### Table 1.1.: We summarize the main features of different \( \mathcal{H} \)-Matrix methods/codes for dense matrices. “\( \text{MATRIX} \)” 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 \( K_{ij} \). “\( \text{LOW-RANK} \)” indicates the method used for the off-diagonal low-rank approximations: “\( \text{EXP} \)” indicates kernel function-dependent analytic expansions; “\( \text{EQU} \)” indicates the use of equivalent points (restricted to low \( d \) problems); “\( \text{ALG} \)” indicates an algebraic method. “\( \text{PERM} \)” indicates the permutation scheme used for dense matrices: “\( \text{OCTREE} \)” indicates that the scheme doesn’t generalize to high dimensions; “\( \text{NONE} \)” indicates that the input lexicographic order is used; and “\( \text{TREE} \)” indicates geometric partitioning that scales to high dimensions. \( S \) indicates whether a sparse correction (FMM or \( \mathcal{H}^2 \)) is supported. We present comparisons with \( \text{ASKIT} \), \( \text{STRUMPACK} \), and \( \text{HODLR} \).

<table>
<thead>
<tr>
<th>METHOD</th>
<th>MATRIX</th>
<th>LOW-RANK</th>
<th>PERM</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMM [16]</td>
<td>( K(x_i, x_j) )</td>
<td>EXP</td>
<td>OCTREE</td>
<td>Y</td>
</tr>
<tr>
<td>KIFMM [66]</td>
<td>( K(x_i, x_j) )</td>
<td>EQU</td>
<td>OCTREE</td>
<td>Y</td>
</tr>
<tr>
<td>BBFMM [24]</td>
<td>( K(x_i, x_j) )</td>
<td>EQU</td>
<td>OCTREE</td>
<td>Y</td>
</tr>
<tr>
<td>HODLR [5]</td>
<td>( K_{ij} )</td>
<td>ALG</td>
<td>NONE</td>
<td>N</td>
</tr>
<tr>
<td>STRUMPACK [60]</td>
<td>( K_{ij} )</td>
<td>ALG</td>
<td>NONE</td>
<td>N</td>
</tr>
<tr>
<td>ASKIT [53]</td>
<td>( K(x_i, x_j) )</td>
<td>ALG</td>
<td>TREE</td>
<td>Y</td>
</tr>
<tr>
<td>MLPACK [21]</td>
<td>( K(x_i, x_j) )</td>
<td>EQU</td>
<td>TREE</td>
<td>Y</td>
</tr>
<tr>
<td>GOFMM</td>
<td>( K_{ij} )</td>
<td>ALG</td>
<td>TREE</td>
<td>Y</td>
</tr>
</tbody>
</table>

**\( \mathcal{H} \)-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 [36, 9, 3, 32, 41, 65].

Let us briefly summarize the \( \mathcal{H} \)-Matrix classification. Recall the decomposition \( K = D + UV + S \) (1.1). If \( S \) is zero the approximation is called a hierarchical 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 hierarchical 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 \( N \) [15] and the complexity bounds are no longer valid. Recently, algorithms have been published to effectively compress FMM and \( \mathcal{H}^2 \)-matrices [20, 67]. One of the most scalable methods is \( \text{STRUMPACK} [26, 60, 55] \), which constructs an HSS approximation of a square matrix (not necessarily SPD) and then uses it to construct an approximate factorization. For dense matrices \( \text{STRUMPACK} \) uses the lexicographic ordering. If no fast matrix-vector multiplication is available, \( \text{STRUMPACK} \) requires \( O(N^2) \) work for compressing a dense SPD matrix, and \( O(N) \) work for the matvec. ¹

¹This section is extracted from the introduction, written by G.Biros, of [17], a work in collaboration with C. Yu, J. Levitt and G. Biros.
2. Theory

In this chapter we give a brief and informal description of necessary mathematical methods used for the topic of hierarchical approximations for symmetric positive definite matrices. It can be seen as broad review of literature on the subject.

This chapter starts with explaining low-rank approximation techniques for general matrices. We then outline the concept of hierarchical representations of matrices. The chapter continues by explaining memory- and runtime-efficient arithmetic operations using the concept of hierarchical approximations.

Readers that are familiar with algebraic fast multipole methods are advised to skip this chapter.

2.1. Low-rank approximations

In this section we begin (2.1.1) with a motivation related to $N$-body problems, which forms the ground for ASKIT: Approximate kernel independent treecode. Our approach is an extension of ASKIT but nevertheless its idea is sparked by the $N$-body problem. In (2.1.2) we cover text-book methods for algebraic compression algorithms. In (2.1.3) we introduce tools from randomized linear algebra which serve as a reference to our approach.

2.1.1. Geometric motivation from $N$-body problems

$N$-body methods are used in many disciplines: simulations of gravity and Coulomb potentials, waves and scattering or fluids and transport; in data analysis the same problem occurs in machine learning, geostatistics and image analysis.

Given a set of $N$ targets $x_i \in \mathbb{R}^d$, a set of $N$ sources $x_j \in \mathbb{R}^d$, weights $w_j$ and a kernel function $K(x_i, x_j)$ (some pairwise potential), we want to compute a target potential $u_i$,

$$u_i = u(x_i) = \sum_{j=1}^{N} K(x_i, x_j)w_j$$

This operation can be viewed as a matrix-vector product with complexity $O(N^2)$ and is often the computational bottleneck of such simulations.

We want to reduce this complexity by exploiting low rank blocks in matrix $K$ ($K_{ij} = K(x_i, x_j)$) using a hierarchical data structure. We rewrite

$$u_i = \sum_{p \in \text{Near}_i} K_{ip}w_p + \sum_{p \in \text{Far}_i} K_{ip}w_p$$

where $\text{Near}_i$ is a set of near points, whose contributions need to be regarded individually and $\text{Far}_i$, being the set of far points, whose contributions can sufficiently be computed by a low-rank approximation.
2. Theory

Figure 2.1.: The curse of dimensionality. On the left: sample set plotted in one dimension with 11 points in gray area (unit bin). Middle: in two dimensions with 6 points in unit quad. Right: in unit cube (three dimensions) lie only 4 points. In order to have at least \( m \) data points in a leaf where its tree is split in all its underlying dimensions, it requires \( m \cdot 2^d \) data points. Figure taken from [59].

Hierarchical interaction calculation algorithm

The idea of such approximations relies on the observation that an interaction force with respect to a far-away individual point may be neglected; however, the force with respect to a far-away (we later cover near-far pruning criteria) cluster may not be neglected. Thus, we want to group far-away points in clusters hierarchically. In this way, we will calculate for a single point \( i \) some interactions from near points \( j_{\text{near}} \) individually, but interactions from far clusters \( \{\text{Far}_j \mid j \in \text{Far}, x_j\} \) only approximately (we aggregate weights at skeletons or center of mass of clusters).

In terms of celestial simulations one can assume that we represent a clustered formation of stars (or solar systems) in terms of its accumulated mass located at its center of mass. For points (planets/stars) in different clusters we approximate the incoming gravitational force from that cluster by its aggregated mass e.g. at its center. In this context this represents a pseudo-point, since we do not necessarily have a data point located at the center; alternatively we can use (one or more) actual data points serving as “skeletons” at which we aggregate the mass of its underlying cluster.

Hierarchical domain decomposition

The most naive hierarchical domain decomposition is a geometrical equidistant splitting. In two dimensions one can imagine a quadratic surface, that is cut into four equally sized subcells – which can be represented in a quad-tree data structure. This becomes infeasible in high dimensions, as number of children are \( 2^d \). A small example is given in Figure 2.1.

In this work we desire the following properties:

- binary tree,
- (roughly) same node sizes,
- adaptive to dataset of intrinsic low dimension.
Figure 2.2.: Circles denote points in the data set; points \(q_i\) are query points for tree reference (Figure 2.3). If cells containing e.g. \(q_i\) have small diameter (max. distance between data points of same leaf), then these points are expected to have similar properties. Left: space partitioning in \(\mathbb{R}^2\) by a \(kd\)-tree of three levels. Right: projection tree/ball tree.

A \(k\)-\(d\) (\(k\)-dimensional tree) tree partitions a \(k\)-dimensional space hierarchically by selecting one coordinate direction and splitting the dataset into partitions with (roughly) equal number of points. In a geometrical sense this kind of partitioning corresponds to hyper-rectangular cells, see Figure 2.2. In order to divide a dataset of 100 dimensions into hyper-rectangular with size (side-lengths) of at least half of the domain size, we would need a tree of level at least 100. There are \(2^{\text{level}}\) leaves in a binary tree, so we also require \(2^{100}\) datapoints (assuming we want a fully occupied tree).

In order to overcome this curse of dimensionality a widely used approach in data analysis is a projection tree or a ball tree; their main feature is that it automatically adapts to datasets of low intrinsic dimension. At this point we neglect whether the projection axis is chosen deterministically or at random. Instead of splitting along one coordinate dimension (\(k\)-\(d\) tree) we split along a \((d - 1)\)-dimensional plane. The axis of projection can be chosen in a way such that this projection tree is similar to a ball tree. Due to the nature of the projection there is no dependence on the external dimension \(d\) anymore, but only on the (hopefully) low intrinsic dimension of the dataset. We have visualized this on the right in Figure 2.2. [22]

Morton IDs

Morton IDs are an efficient way to traverse in tree data structures. It is a bit array that encodes the position of a node in a tree.

It encodes the path from root to the tree node: entries are zero if traversed left, and one if traversed right. [48]

For \(q_1\) in Figure 2.3 the Morton ID reads: 000 and for \(q_2\) 111.
2. Theory

Figure 2.3.: Binary tree corresponding to Figure 2.2. Note the numbering (i.e. which child is left); in human-perceptible dimensions this can be intuitive: first priority who’s more left (x-axis), and second who’s higher (y-axis). In higher dimensions this depends on the direction of the projection axis. Note that interior nodes correspond to pseudo points, and for visualization here (Figure 2.2) chosen to be (roughly) in the middle between split points.

2.1.2. Algebraic compression algorithms

Let us look at the earlier mentioned matrix-vector product

\[ u = Kw \]

where \( u \in \mathbb{R}^N \) are called potentials, \( K \in \mathbb{R}^{N \times N} \) the system matrix and \( w \in \mathbb{R}^N \) the weights.

We wish to approximate \( K \) by

\[ K \approx \tilde{K} := D + U + S \]

where \( D \) is a diagonal block matrix representing close-by points, \( U \) a dense block matrix representing off-diagonals (which allows a block-wise low rank approximation) and \( S \) a sparse matrix, representing points in off-diagonals, that are nevertheless regarded as important such that they need to be evaluated directly. We hence write

\[ u \approx (D + U + S)w \]

Furthermore we wish that \( \tilde{K} \) has less dependent information and thus, reduces storage and speeds up arithmetic operations. Any algorithm that computes \( \tilde{K} \) from \( K \) can be referred to as a compression algorithm. [28, 9]

Similarly we can use this approach in computing a solution of \( N \) linear algebraic equations

\[ Kx = b \]

where \( K \) is dense and nonsingular, \( b \in \mathbb{C}^N \) given and \( x \) a \( N \)-dimensional solution vector. One approach is to calculate a pseudo-inverse solution by a compressed \( \tilde{K} \); the other approach can use a fast matrix-vector product in iterative solution methods (Arnoldi-iteration [7], conjugate gradient [62], GMRES [61] etc.). The number of iterations in the
2.1. Low-rank approximations

solver depends on the condition of the matrix; preconditioning is indispensable, but for certain problems finding a suitable preconditioner can be difficult. In general a bottleneck of iterative solvers lies in memory- and runtime-expensive matrix-vector multiplications. \cite{28, 5}

In general, these ideas of matrix compression are based on the observation that matrices can often not be represented by global low-rank, but it is reported that for many model problems some blocks can be compressed with a low-rank (see subsection 2.2.3).

**Reduced singular value decomposition**

A very common way to compute a low-rank approximation of a matrix $A \in \mathbb{C}^{m \times n}$ is a singular value decomposition (SVD). Since we mostly encounter “narrow” matrices, i.e. $n << m$, we want to discuss a reduced form of the SVD here. We are seeking a decomposition

$$A = U \Sigma V^*$$

where $U \in \mathbb{C}^{m \times n}$, $V \in \mathbb{C}^{n \times n}$ are unitary matrices and $\Sigma \in \mathbb{R}^{m \times n}$ is a non-negative diagonal matrix. Its entries ($\Sigma_{ii}$) are called the singular values of $A$ and are in fact the non-negative square roots of eigenvalues to $AA^*$ or $A^*A$. A regular singular value decomposition can be calculated by successive Householder transformations on $A$ using an intermediate bidiagonal matrix $J \in \mathbb{R}^{m \times n}$ and a subsequent diagonalization of $J$. For a reduced “Thin” version we usually use a QR decomposition of $A$. Overall, the computation of the singular value decomposition requires $O(mn^2)$ operations (where $m \geq n$). \cite{27, 9}

Using this approach we can approximate for a $q \times q$ block of matrix $K \in \mathbb{R}^{n \times n}$, its off-diagonal, let us call it $G \in \mathbb{R}^{(n-q) \times q}$. This leads among other matrices to $U \in \mathbb{R}^{n \times n}$. Assuming that $n$ is large (and $q$ is fairly small) this approach becomes very unfavorable in terms of memory. (We need to save the basis for the decomposition.)

**Interpolative decomposition**

Roughly speaking, in the interpolative decomposition, given a matrix $G$ of rank $k$ we find $k$ of its columns in such a fashion that they form a well-conditioned basis for the remaining columns.

This approach gives us benefits in terms of memory storage and saves operations leading to a complexity of $O(kmn)$. For any matrix $G \in \mathbb{R}^{m \times n}$ of rank $k$ there exists a subset of columns from $G$ building $G_{col} \in \mathbb{R}^{m \times k}$ and a projection matrix $P \in \mathbb{R}^{k \times n}$, such that

$$G_{col}P = G$$

Additionally, some subset of $P$ consists of the $k \times k$ identity matrix and $P$ is not too large.\cite{45}

Let us suppose the rank of the matrix $G$ is $k > s$, and $k \leq n, m$. We can find a $G_{col} \in \mathbb{R}^{m \times s}$ and $P \in \mathbb{R}^{s \times n}$, such that

$$G_{col}P \approx G$$

and can estimate the error as

$$\|G_{col}P - G\|_2 \leq \sigma_{s+1} \sqrt{s(n-s) + 1}$$
2. Theory

where $\sigma_{s+1}$ is the $(s+1)$st singular value of $G$. [45]

An interpolative decomposition can be calculated using a rank revealing QR decomposition.[9]

We decompose $G$ into

$$G\Pi = QR$$

where $Q \in \mathbb{C}^{m \times n}$ has orthonormal columns, $R \in \mathbb{R}^{n \times n}$ is upper triangular and $\Pi \in \mathbb{R}^{n \times n}$ is a permutation matrix from a pivoted QR. We write in split form

$$G\Pi = QR = [Q_{\text{left}} \hspace{1em} Q_{\text{right}}] \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \approx [Q_{\text{left}} \hspace{1em} Q_{\text{right}}] \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

where $Q_{\text{left}} \in \mathbb{R}^{n \times s}, R_{11} \in \mathbb{R}^{s \times s}$ (and appropriate other sizes), whereas we can guarantee that $\|R_{22}\| = \mathcal{O}(\sigma_{s+1})$. We approximate by disregarding $R_{22}$ and simplify to

$$G\Pi \approx [Q_{\text{left}}R_{11} \hspace{1em} Q_{\text{left}}R_{12}] = Q_{\text{left}} \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

We now write

$$G_{\text{col}} = Q_{\text{left}}R_{11}$$

where we term $G_{\text{col}}$ as the first $s$ columns of $G\Pi$; since $\Pi$ represents a permutation matrix $G_{\text{col}}$ turns out to be a subset of columns of $G$. We refer to the source points which correspond to the columns $G_{\text{col}}$ as the skeleton points to this node. Using the approximation assumption

$$G_{\text{col}}P \approx G\Pi$$

we rewrite

$$Q_{\text{left}}R_{11}P = Q_{\text{left}} \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

and simplify this expression to

$$R_{11}P = \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

It remains to solve a system of linear of equations for $P \in \mathbb{R}^{s \times n}$; it appears that $P$ involves an $s \times s$ identity matrix, namely (in MATLAB notation)

$$P = \begin{bmatrix} Id_{s \times s} & R_{12} \backslash R_{11} \end{bmatrix}.$$  

In conclusion, we calculate a low rank factorization of a matrix block using the interpolative decomposition; we have bounded the error in terms of the $(s+1)$st singular value. However, those rank-revealing factorizations are less reliable than the SVD. [9]

A very neat and asymptotically more efficient algorithm that involves randomized linear algebra is described here [45]. It is used in the software framework STRUMPACK [60].

2.1.3. Randomized linear algebra

Randomized algorithms have become a useful tool in numerical analysis, data analysis applications, and in a wide range of scientific computing.

In this section we selectively explain row sampling from matrices, which is used in ASKIT (and GOFMM). We also explain the concept of leverage scores that describe an optimal choice for sampling (but are too expensive to compute).
2.2. Hierarchical matrices

Sampling matrix rows

In this context one wants to avoid computing low-rank factorizations of big blocks of matrices. Introducing randomized sampling originates from the observation that a small set of rows/columns can be used to describe most of the relevant contributions in a numerically low-rank matrix. \[40, 49\]

As introductory example let us look at a matrix $G \in \mathbb{R}^{n \times n}$ with $n$ rows, where the first $n - 1$ rows are parallel denoted as row $r_1 \in \mathbb{R}^n$ (can be scaled $g_1 \ldots n \in \mathbb{R}$) and the last row $r_2 \in \mathbb{R}^n$ is orthogonal to them.

\[
G = \begin{bmatrix}
g_1 \ast r_1 \\
g_2 \ast r_1 \\
\vdots \\
g_{n-1} \ast r_1 \\
r_2
\end{bmatrix}
\]

In order to capture the row space of $G$, one needs at least one sample from $r_1$ and one from $r_2$. With a uniform sampling scheme, this requires $O(n)$ row samples.

Leverage scores

Statistical leverage scores describe a measure for the importance of a row to the remaining contributions in the row space. Leverage scores can be computed using the matrix of left singular vectors to $G$, often denoted as $U$. The statistical leverage scores for row $j$ in $G$ to a numerical rank $s$ are then defined as

\[
l_j = \| U(j, 1 : s) \| .
\]

In the example, the leverage score for the last row $r_2$, will be high. Thus, we could sample rows from probabilities according to their leverage score and we will accurately capture the row space with two samples. However, the computation or estimation of leverage scores requires to “touch” all entries of the block and is too expensive in practice; sampling uniformly is cheap, but may not be useful for general matrices. \[46, 49, 25\]

As a greedy heuristic, ASKIT uses nearest-neighbor information to sample row targets according to their importance. Suppose we have a well-defined physical problem domain and the kernel is a decreasing function of distance; rows corresponding to nearest neighbors to that block are expected to have a large norm, and loosely speaking also have a large leverage score. Note, that nearest neighbors can be approximately computed without touching all entries of the matrix. \[49\]

2.2. Hierarchical matrices

Given a system of $N$ equations the optimal efficiency in computational work is $O(N)$. In many situations we can derive a sparse system matrix (where we know optimal solution algorithms), but problems arise for non-sparse matrices. Given $N^2$ entries in the system matrix it seems to be unavoidable to have algorithms of less than complexity $O(N^2)$; cases
2. Theory

like the Fast Fourier Transform (FFT) show that this does not hold in general. We here want
to describe the class of $\mathcal{H}$-matrices (where $\mathcal{H}$ abbreviates hierarchical) which are data-sparse
meaning that these matrices can be described by only few data needed for their representa-
tion. As a result we want to perform an approximative matrix-vector multiplication in
almost linear complexity as well as a matrix factorization. For approximate evaluation
we can achieve (asymptotic) linear complexity using a two-sided compression and denote
matrices as $\mathcal{H}^2$. [35]

In analytic fast multipole methods used for $N$-Body problems, Taylor or kernel-dependent
function expansions are defined to approximate the far-field [33]. Semi-analytic approaches
select equivalent source points using the density distribution of the data and build analytic
outgoing representations that are accurate at seeded points e.g. on a bounding sphere.
This approach is implemented in a black-box fast multipole method; however, the effort
increases significantly in high dimensions because seeded and selected points grow expo-
nentially [49, 57, 66, 48]

$\mathcal{H}$-matrices and skeletonizations can be seen as algebraic generalizations from those
methods or derived through the not-so-popular panel clustering method [37, 48, 49]. Al-
ternatively, well-structured $\mathcal{H}$-matrices can be generated with wavelet compression tech-
niques [2, 9].

2.2.1. SPD matrix as gramian

Any symmetric positive-definite (SPD) matrix can also be described as a Gramian matrix
of some set of vectors $\phi$. Given a matrix $K \in \mathbb{R}^{N \times N}$ (e.g. from a kernel function) being
symmetric positive definite, we can assume that there exists a set of vectors $\phi_i \in \mathcal{V}$ which
define this matrix by a scalar product,

$$K_{ij} = \langle \phi_i, \phi_j \rangle$$

Note that we do not have direct access to the underlying vector set of $\phi_i$, but only to the
scalar product between each other. The scalar product $\langle \phi_i, \phi_j \rangle$ can be viewed as projection
of $\phi_i$ onto $\phi_j$.

As a side note, a very naive and limited way in calculating a set of finite-dimensional
$\phi_i \in \mathbb{R}^N$ is the Cholesky decomposition, i.e. $K = LL^*$ where $L$ is lower triangular. This
possible set of $\phi_i$ are the columns of $L$. Every Hermitian positive definite matrix (and
thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky
decomposition; however, by having $L$ lower triangular we restrict that $\phi_i$ are constructed
in a limited manner; without loss of generality the Gramian set $\phi_i$ is not unique and may
also be composed in a different space $\phi_i \in \mathcal{V} \not\subset \mathbb{R}^N$.

2.2.2. Hierarchical matrix partitioning

It remains very difficult to explain and justify the assumption that block low-rank approx-
imations do exist for some application problems (despite the fact that, for practical reasons
it seems intuitive). Accepting that assumption that large coefficient matrices can be partitioned
in such a way that most of their blocks are nearly low-rank matrices, questions still
arise how an optimal index-ordering can be achieved.
2.2. Hierarchical matrices

The problem reads: given a set of indices \( I \) we want to find an index order of disjoint subsets \( I_a, I_b \subset I \) (\( I_a \cap I_b = \emptyset, I_a \cup I_b = I \)) such that the off-diagonal blocks can be approximated by a low rank

\[
\text{minimize } \text{rank}(A[I_a, I_b] - S)
\]

where \( S \) can be some sparse matrix also subject to optimization. \( A[.,.] \) is an appropriate sub-block of system matrix \( A \), and \( S \in \mathbb{R}^{n \times m} \), where \( n \) is the number of elements in \( I_a \), \( m \) in \( I_b \); in the easiest case \( n, m = \frac{N}{2} \), where \( N \) is the number of elements in \( I \).

For certain application fields optimal orderings may be derived; but clearly to serve as a general black-box algorithm we need to find a greedy heuristic.

Partitioning in physical space

This partitioning is based on the assumption that near points have a big influence, whereas the physical far-field can be approximated sufficiently. This problem is subject to the discipline of subspace clustering and it can be associated with section 2.1.1. Different techniques can be found:

- **Quad-tree splitting and space filling curve numbering**
  For boundary integral equations for physical problems a hierarchical subdivision structure can be described by a space tree. A well-suited numbering then corresponds to a hierarchically refined space filling curve (e.g. \( z \)-curve in \( 2d \)). \([19]\)

- **Geodesic distance metric**
  Geodesic distance can loosely be described as the distance of points connected with a geometric structure. For boundary value problems there are advantages in this metric versus the Euclidean distance \([35]\).

- **High dimensional subspace clustering**
  As described in section 2.1.1 a projection/ball tree is a suitable choice for data sets of high physical, but low intrinsic dimension. The ordering can be determined by a pre-order tree traversal; it may be challenging to draw this numbering scheme as space filling curve. Note that this scheme is used in \textsc{AskIt}. \([49, 22]\)

Note that knowledge of the physical domain additionally helps in the far-field pruning criterion. The mentioned sparse matrix \( S \) represents contributions that are additionally evaluated directly and cannot be approximated by a low-rank. A distance based pruning criterion is used in classical \( N \)-body force calculation algorithms (Barnes-Hut .. \([8]\)) up to three dimensions. In higher dimensions data will lie in roughly similar Euclidean distances (concentration of measure effect), and hence this distance metric becomes an impracticable criterion \([23]\). For this reason, \textsc{AskIt} uses neighbor-based pruning \([49]\).

In general, most hierarchical matrix codes require points in physical space \([12, 26, 14, 49, 4]\).

Black box geometry-free

If we neglect computational complexity, one can think of a brute force optimization algorithm. It will be very hard to calculate the exact solution, but we could relax on that
2. Theory

criterion (i.e. searching approximate optimal solution). The algorithm, as simple as it could be: guess a partitioning, try and repeat. To bring this back to reasonable compute effort, some kind of greedy heuristic will be necessary.

A black-box geometry-free partitioning has been developed for a sparse system matrix in elliptic boundary problems. It can be seen as a graph reordering algorithm or more generally a graph bisection algorithm. [30] To our knowledge there exists no technique for general dense matrices.

2.2.3. Hierarchical off diagonal low rank matrices

Classical $\mathcal{H}$-matrix approach

Assuming we have a suitable partitioning we would like to express a matrix $K \in \mathbb{R}^{N \times N}$ in hierarchical blocks. In classical $\mathcal{H}$-Matrices this corresponds to the following block structure

$$K = \begin{bmatrix} K^{(1)}_{11} & K^{(1)}_{12} \\ K^{(1)}_{21} & K^{(1)}_{22} \end{bmatrix} = \begin{bmatrix} K^{(1)}_{11} & (U\Sigma V^*)_{12}^{(1)} \\ (U\Sigma V^*)_{21}^{(1)} & K^{(1)}_{22} \end{bmatrix}$$

$$= \begin{bmatrix} K^{(2)}_{11} & (U\Sigma V^*)_{21}^{(2)} \\ (U\Sigma V^*)_{21}^{(2)} & K^{(2)}_{22} \end{bmatrix} = \begin{bmatrix} K^{(3)}_{33} & (U\Sigma V^*)_{34}^{(2)} \\ (U\Sigma V^*)_{34}^{(3)} & K^{(4)}_{44} \end{bmatrix}$$

where superscript indexes denote the level. Note that in this representation we ignore a possible sparse matrix $S$ responsible for additional direct evaluations. [5, 65, 26]

Assume now we multiply this compressed matrix with a vector $w$, namely $u = Kw$. If approximation $(U\Sigma V^*)_{12}^{(1)}$ is inaccurate, the error of the first $\frac{N}{2}$ is dominated by this inaccuracy, and there is no possibility of refinement (for the influence of block $(U\Sigma V^*)_{12}^{(1)}$).

Skeletonization approach

In a slightly modified algorithm we want to approximate the outgoing contribution from sources to desired target points – possibly all. This approach is outlined in Figure 2.4. Recall the multiplication with a vector $u = Kw$; if the approximation of block $G^{(1)}_{2}$ is deemed inaccurate, one can refine in evaluation to the representation of $G^{(2)}_{3,4}$. The accuracy of $w_i$ ($0 \leq i < \frac{N}{2}$) then depends on the accuracy of $G^{(2)}_{3,4}$ towards targets (rows) $i$.

One may now wonder where the hierarchical approach influences this method. In ASKIT the construction of the low-rank decomposition of block $G^{(1)}_{2}$ will be built on a prior result from $G^{(2)}_{3,4}$. In detail, appropriate “skeletons” will be selected on lower levels, which correspond to columns in the matrix. The low-rank decomposition of block $G^{(1)}_{2}$ will thus only be computed on a subset of columns merged from the level below, hierarchically created by a “bottom-up” approach.

Although these approximations may be very accurate and compact, their construction is much too expensive. The sampling of targets (where the approximation should be accurate) is the subject of subsection 3.2.3.
2.3. Arithmetic operations

We begin (2.3.1) this section with an computational operation count for a classical hierarchical matrix-vector product. It is asymptotically similar to the skeletonization approach. We then (2.3.2) explain a fast multipole like evaluation scheme which asymptotically reduces necessary operations to $O(N)$. In (2.3.3) we shortly describe a matrix factorization method as reference (which our code does not yet offer).

2.3.1. Matrix vector product

Classical $H$-matrix

We refer to memory requirements or floating point operations interchangeably in the following as Cost. Suppose we multiply

$$u = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

where we have decomposed diagonals hierarchically and off-diagonals with

$$K_{12} = G_{col} P$$

where $K \in \mathbb{R}^{n \times m}$, $G_{col} \in \mathbb{R}^{n \times k}$ and $P \in \mathbb{R}^{k \times m}$. As explained in subsection 2.1.2 this representation is suitable in memory as $\text{Cost}_{\text{storage}} = k \cdot (n + m) = O(n + m)$. Let us express the matrix-vector multiplication block-wise. We need to calculate $K_{11}w_1$, $K_{12}w_1$, $K_{21}w_2$, $K_{22}w_2$ and add respective terms. We count the operation for off-diagonals;
Figure 2.5.: For each node we here select two skeletons; on interior levels this is collected from children’s skeletons. Weights are aggregated at skeletons from bottom up, corresponding to \( w = P[w_{\text{child}}, w_{\text{child}}]^T \). In literature this is step is often called multipole to multipole translation. Left: leaf level \( l = 3 \) in green; Middle: interior level \( l = 2 \) in red; Right: upper level \( l = 1 \) drawn in blue.

\([P]_{12}w_1\) gives on each row \((k\cdot)\) a scalar product; multiply and add to a sum, \(2m-1\) operations.

Now we multiply the intermediate vector \( Pw \) of size \( k \) this with \([G_{\text{col}}]_{12}\), which leads to \( n \cdot (2k - 1) \) operations. Let us say \( n, m = N = \frac{N}{4} \), \( \text{Cost}_{\text{diag}} = k(2N - 1) - \frac{N}{4} \).

For the overall product we can write the recurrence equation with \( N = 2^l \), (note we included +\( N \) for adding),

\[
\text{Cost}_{\text{MatVec}}(l) = 2 \cdot \text{Cost}_{\text{MatVec}}(l-1) + k \cdot (4N-2)
\]

Applying \( \text{Cost}_{\text{MatVec}}(0) = 1 \), we get

\[
\text{Cost}_{\text{MatVec}}(l) = 4l \cdot k \cdot N - k \cdot N + 2.
\]

Note that on low levels a constant rank \( k \) is unjustified; however, for simplicity this will give us an upper bound. As a result, \( \text{Cost}_{\text{MatVec}} = \mathcal{O}(N \cdot \log_2(N)) \).

**Skeletonization approach**

As mentioned earlier, in skeletonization approaches the approximated matrix \( K_{12} \) is much smaller than \( \frac{N}{4} \times \frac{N}{4} \), i.e. it only assumes skeleton source points. In each level we aggregate \( w = P \left[ w_{\text{child}}, w_{\text{child}} \right]^T \), see Figure 2.5.

Assuming a fixed skeletonization size \( s \), this step involves roughly \( \mathcal{O}(s^2) \) operations per level; on leaf level we need to touch all points, and cannot do better than \( \mathcal{O}(N) \).

Let’s evaluate the \texttt{matvec} for a single point \( u_i \). We aggregate incoming representations from each tree level using a scalar product, i.e. \( C = 2 \cdot s - 1 \) operations. Corresponding to Figure 2.5, from the blue level, red level, green level individually. Assuming \( \log(N) \) tree levels, this will give us a cost of \( \mathcal{O}(\log(N)) \). For evaluating all points with this single evaluation scheme we end up with \( \mathcal{O}(N \log(N)) \) operations.
2.3.2. FMM-like evaluations

The idea of the fast multipole method (FMM) in this context is that we approximate the incoming representation as well. Let us look at Figure 2.5 again. In the earlier evaluation scheme we calculated for all points in the upper part of the tree the incoming contributions from the blue skeletons in the right picture ($\frac{N^2}{2} \cdot 2$ operations). The idea now is that we perform a “skeleton to local” translation. We only calculate the incoming representation of the blue skeletons on corresponding blue skeletons of the upper node (not drawn). From these “blue skeletons” we then approximate the incoming representation on the red skeletons (by additionally adding the contribution from the sibling).

There are more elaborate forms for better accuracy with more flexible pruning criteria, but we want to keep things simple here. This downward-pass basically calculates a projection from its skeletons to its children’s skeletons, i.e. $[u_{\text{child}}, u_{\text{child}}]^T = P^T u$. Spoken in the picture from blue to red, this corresponds to the incoming force from the upper node. Additionally, one needs to add the contribution from its sibling.

To evaluate one point’s potential on a leaf level, one only needs the projection from leaf node’s skeleton potential to itself, $O(s)$. To evaluate potentials for all $N$ points, we only spend $O(N)$ operations.

2.3.3. Matrix inversion

Inversion algorithms for hierarchical matrices rely on the Sherman-Morrison-Woodbury (SMW) formula. We want to calculate the inverse of a matrix $K \in \mathbb{R}^{N \times N}$, which we already have in compressed form

$$K = D + UV = D(Id + D^{-1}UV) = D(Id + WV)$$

where $W = D^{-1}U$. We can express the inverse as

$$K^{-1} = (Id + WV)^{-1}D^{-1} = (Id - W(Id + VW)^{-1}V)D^{-1} = (Id - WZV)D^{-1}$$

where $Z = (Id + VW)^{-1}$. Note that the intermediate step two is applied according to the SMW formula. Having a compression $K = D + UV$ already, we now only need to calculate $W$ and $Z$ in every node. [5, 18, 38]

2.4. Related work

This idea of far-field approximations was first described by Barnes & Hut [8], extended by Greengard & Rokhlin [33] to so called “fast multipole methods” and forms the backbone of the software release by George Biros’ group ASKIT: Approximate Skeletonization Kernel-Independent Treecode. Many FMM codes and also ASKIT require points in the original space and the underlying kernel for building a treecode according to geometric partitioning. Thus, those algorithms are very domain specific and hence, it can be cumbersome to integrate it in existing research codes. The idea of this project is to design a “black-box” algebraic algorithm, where no distances in the original space will be needed.
A collection of template algorithms for hierarchical treecodes have been developed in order to generalize research efforts on various sides.\cite{14} A few packages exist for hierarchical off-diagonal low-rank matrices optimized for distributed-memory systems. However, they don’t assume anything about how the matrix is created, but assume that off-diagonal blocks are low rank \cite{26, 60, 5}. There has been some work on numbering using graph partitioning for sparse matrices from discretization of certain problems \cite{30}. Other software packages that use low-rank approximation techniques include: Hlib (for $\mathcal{H}$ and $\mathcal{H}^2$ matrices) \cite{12} and MUMPS (sparse direct solver with Block Low-Rank approximation techniques) \cite{6}.
3. Methods

Note that our approach is very similar and heavily inspired by ASKIT: Approximate Skeletonization Kernel-Independent Treecode.

In this chapter we (3.1) cover a geometry-free tree construction and define a notion for distance between rows. We then (3.2) describe the compression algorithm that uses an adaptive approximation rank and importance sampling. Next (3.3), we point out the steps necessary in the evaluation phase of the algorithm, as well as the parallel approach. Lastly (3.5), we explain the numerical setup and the test cases.

3.1. Geometry-oblivious techniques

In this project we applied a black box geometry-oblivious heuristic to define a distance between two rows (and columns). This allows

1. a hierarchical binary partitioning of the matrix,
2. constructing neighbors for target sampling and pruning.

We start this section with classical partitioning algorithms in physical (or feature) space, and then extend this approach for the geometry-oblivious case using two strategies. We then outline the algorithm that is applied for hierarchical splitting, which is similar to a metric ball tree. Later, we explain a nearest neighbor search based on random projection trees that is applicable for defined distance metrics.

3.1.1. Matrix reordering

The numerical rank of off-diagonal blocks heavily depends on suitable reordering. First, we wish that we can compute a low approximation rank (to allow significant compression). Second, we wish that we can approximate outgoing representations of clusters with a neighbor-based sampling approach.

Geometric

If we are given points \( \{ x_i \}_{i=1}^{N} \in \mathbb{R}^d \), then we can use the geometric \( \ell^2 \)-distance. This means, we build a hierarchical subdividing tree splitting such that we cluster close points, i.e. the distance \( d_{ij} = \| x_i - x_j \|_2 \) between \( i \) and \( j \) of the node is minimized. This approach is used in ASKIT [49].
3. Methods

Gram $\ell^2$

We consider a given symmetric positive-definite (SPD) matrix $K$ in terms of an underlying Gramian set of vectors as $K_{ij} = K(x_i, x_j) = \langle \phi_i, \phi_j \rangle$. Note that we do not have direct access to the underlying vector set of $\phi$, but only to the inner product between the vectors.

First, we look at the $\ell^2$ distance in Gramian space, conceptually similar to a geometric splitting. We can express the distance between rows in terms of Gram vectors, i.e.

$$d_{ij}^2 := \| \phi_i - \phi_j \|^2 = \langle \phi_i - \phi_j, \phi_i - \phi_j \rangle = \langle \phi_i, \phi_i \rangle - 2 \langle \phi_i, \phi_j \rangle + \langle \phi_j, \phi_j \rangle$$

and hence

$$d_{ij}^2 := K_{ii} + K_{jj} - 2K_{ij}.$$ 

Gram angles

Here we derive a metric depending on the angle between Gram vectors, i.e. $\langle \phi_i, \phi_j \rangle$.

Although this may appear arbitrary, the idea is sparked by an observation relating the rank of an off-diagonal block to the degree of orthogonality between sets of Gram vectors. Consider disjoint index sets $\alpha, \beta \subset I$ and the corresponding matrix of interactions $K_{\alpha\beta}$. If we define $\phi_\alpha$ to be the matrix with columns $\{ \phi_i \}_{i \in \alpha}$ and define $\{ \phi_j \}_{j \in \beta}$ similarly, then $K_{\alpha\beta} = \phi_\alpha^T \phi_\beta$. We may view $K_{\alpha\beta}$ as a projection of $\phi_\beta$ onto the span of $\phi_\alpha$, so the rank of $K_{\alpha\beta}$ is equal to the dimension of the projection. Thus, we require a measure of distance that clusters Gram vectors to form orthogonal subspaces, rather than small volumes as above.

By the law of cosines we write

$$\langle \phi_i, \phi_j \rangle = \| \phi_i \| \cdot \| \phi_j \| \cdot \cos(\langle \phi_i, \phi_j \rangle)$$

and rephrased

$$\cos(\langle \phi_i, \phi_j \rangle) = \frac{\langle \phi_i, \phi_j \rangle}{\| \phi_i \| \cdot \| \phi_j \|} = \frac{\langle \phi_i, \phi_j \rangle}{\sqrt{\| \phi_i \|^2 \cdot \| \phi_j \|^2}} = \frac{\langle \phi_i, \phi_j \rangle}{\sqrt{\langle \phi_i, \phi_i \rangle \langle \phi_j, \phi_j \rangle}}$$

We thus define a distance using a squared sine, $\mathcal{V} \rightarrow [0, 1]$, i.e.

$$d_{ij}^2 := \sin^2(\langle \phi_i, \phi_j \rangle) = 1 - \cos^2(\langle \phi_i, \phi_j \rangle) = 1 - \frac{K_{ij}^2}{K_{ii}K_{jj}}$$

Note that the distance is small if the vectors are nearly collinear (which we consider near) and large (close to one) if the vectors are nearly orthogonal (which we consider far). The splitting scheme is visualized in Figure 3.1.

In conclusion, we wish to build a hierarchical subdivision by these criteria. In the latter two, we only require access to the raw elements of $K$.

Metric ball tree

It would be too expensive to perform an exhaustive search, such that in both partitions the notion of distance is minimized. Instead, we perform a metric ball tree like algorithm
3.1. Geometry-oblivious techniques

Figure 3.1.: Hierarchical tree construction. Points ▲/★ represent Gram vectors (in two dimensions). Vector p is farthest away from the approximate center of mass, k is the point farthest away from p. Tree is built on projection towards this axis: ▲ denote points for the left child, ★ for the right. Left: $\ell^2$-based splitting. Solid line can be seen as $(d-1)$-dimensional hyper-plane orthogonal to dotted direction. Plane is located at the median, such that leaves are of equal size. Right: Angle-based splitting. Note that a fictive cone (solid line) can be constructed with a constant aperture $\alpha$ in $(d-1)$ dimensions.

using a projection axis. We calculate an index $p$, as $\phi_p$ being farthest away from the center of mass $\hat{\phi}$ of corresponding points in the tree node

$$\phi_p = \arg\max_j \|\hat{\phi} - \phi_j\|$$

where we look at the distance to the center $\hat{\phi}$ by

$$\|\hat{\phi} - \phi_j\|^2 = \sum_{i=1}^{N} \|\phi_i - \phi_j\|^2 = \sum_{i=1}^{N} \|\phi_i\|^2 + \|\phi_j\|^2 - 2\sum_{i=1}^{N} <\phi_i, \phi_j>$$

Instead of summing $i$ until $N$ in practice we only compute the distance to an approximate center by summing over a random subset; we find $\phi_p$ by taking the maximum.

Subsequently we find the index $k$ to $\phi_k$ being on the opposite end of the distribution, i.e. being far-most away from $\phi_p$. This step is visualized in Figure 3.1; these points span the dotted line.

Furthermore, to construct a tree we consider a direction spanned by $\phi_p$ and $\phi_k$ and split indices $i$ into two subsets (pivot is the median) by

a. a $\ell^2$ based splitting $<\phi_i, \phi_p - \phi_k>$ or $<x_i, x_p - x_k>$,

b. a $(d-1)$-dimensional two-fold cone spanned in direction $\phi_p - \phi_k$, and its opposite direction $\phi_k - \phi_p$, i.e.$\frac{1}{\|\phi_i\|} <\phi_i, \phi_p - \phi_k>\|$. 


21
3. Methods

Note that this scheme slightly differs from the one proposed in the paper [17], but results turned out to be very similar.

Algorithmically, this process is outlined in pseudo algorithm 1. The leaf node size \( m \) is a user-specified parameter; it controls the error and the runtime since it governs the trade-off between near and far interactions.

**Algorithm 1 Build Hierarchical Structure**

```plaintext
function TREEBUILD ( matrix K )
    p ← Point furthest away from center of mass
    k ← Point furthest away from p
    for \( i ← 1 \) to \( n \) do
        \( proj[i] ← \langle \phi_p − \phi_k, \phi_i \rangle \) OR \( \frac{1}{\|\phi_i\|} \| \langle \phi_i, \phi_p − \phi_k \rangle \| \)
    end for
    [idxLeft , idxRight] ← median-split according to proj
    if \( n < m \) then
        lchild ← treeBuild(K[idxLeft , idxLeft])
        rchild ← treeBuild(K[idxRight , idxRight])
    end if
    return treeNode
end function
```

3.1.2. Neighbor search using randomized trees

We now want to find neighbors \( N \) for each index. From the aforementioned tree we have a rough idea about the spatial distribution. Closest points globally (i.e. neighbors) may lie in the same leaf node; however, especially for points on edges of splittings, they may also lie in other leaves. Simply placing a ball (hyper-sphere) around a point and finding points inside can be very ineffective in high dimensions.

All nearest neighbors can be computed arbitrarily accurate (in high dimensions) using randomized projection methods [22]; we simply build a number of randomized projection trees, and treat points of the same leaf as candidate neighbors.

In each iteration and for each hierarchical split, we select a random direction \( \phi_p − \phi_k \) (\( p \) and \( k \) random here), and split a tree node into two children left and right

a. using a fictive orthogonal hyper-plane defined by the median,

b. or fictive hyper-cones defined by a (roughly) equal-sized split.

An exhaustive neighbor search (brute force) is performed for all candidate neighbors. This algorithm is outlined in 2. Randomized trees can be performed in parallel; one needs to be careful when updating the sets of candidate neighbors.

Neighbors can also be considered as input: for sampling and pruning. It can be computed with any method, and also in the physical domain. If using geometric neighbors, it is however also advisable to build the static tree in physical space. We have also a treeBuild implemented working on coordinates.
3.2. Matrix compression

Additionally, many learning tasks require repeated computations on the same data set with different kernel parameters. In this case, the geometric neighbor information will not change between runs, so precomputing and storing them is an efficient approach.

Algorithm 2 Approximate Nearest Neighbors (rkdt)

```plaintext
function FINDNEARESTNEIGHBORS
candidates[N] ← set() ▷ Each node has a set of candidate neighbors
for ii ← 1 to numberOfRandomTrees do
    randomTree ← build randomized tree
    for i ← 1 to N do
        candidates[i] ← Add all indexes in same randomized leaf
    end for
end for
knnArray ← matrix $k \times N$ ▷ k neighbors exhaustively calculated from candidates
return knn
end function
```

3.2. Matrix compression

3.2.1. Compression algorithm

This section explains how we compute a far-field approximation of a block of source indexes $X_\alpha$ towards target indexes $T_\alpha$. This corresponds to approximating columns $i$ on target points $j$, in as shown in Figure 2.4.

Skeletonization of leaves

In order to skeletonize leaves we build neighbor-based target sampling lists $T_\alpha$ (for rows) from a previous all nearest neighbor calculation (see section subsection 3.2.3). Regarding columns, we use all leaf indexes $X_\alpha$. Skeletonization on leaf levels can be performed in parallel.

Skeletonization of interior points

For target sampling we use a neighbor-based target sampling lists (for rows) $T_\alpha$ by merging sampling neighbor lists from the children (see section subsection 3.2.3). We perform an ID on the children to compute skeletons. On the interior level we merge skeletons of the children to serve as equivalent source points used for the interpolative decomposition, i.e. $X_\alpha = S_{left} \cup S_{right}$. Thus, we can only perform the skeletonization of interior nodes after a skeletonization of the children.

Tree traversal and storage

Skeletonization corresponds to a postorder tree traversal: First skeletonize the children, then myself. However, this applies only to computing the skeletons (using a pivoted QR
3. Methods

factorization). The triangular solve to compute \( P = \begin{bmatrix} I_{d \times s} & R_{12} \setminus R_{11} \end{bmatrix} \) can be computed in any order.

In the interpolative decomposition a subset of columns of \( G \) serves as a basis for the compression. We only need to store the skeleton indexes, and \( P_2 = R_{12} \setminus R_{11} \).

3.2.2. Adaptive rank selection

As explained in section 2.1.2 we perform a compression by a interpolative decomposition using a rank revealing QR factorization. Let us call the block for a tree node \( \alpha \) (consisting of columns \( \mathcal{X}_\alpha \) and rows \( \mathcal{T}_\alpha \)) \( \hat{G} \). We can thus only look at the error of the sampled block, i.e.

\[
\| \hat{G} - \hat{G}_{\text{col}} P \|
\]

We estimate the necessary approximation rank with an estimate of the \( i \)th singular value, i.e.

\[
\hat{\sigma}_i = R_{ii} \left( \frac{q}{q'} \right)^{1/2} \left( \frac{N - q}{l} \right)^{1/2}
\]

where \( q \) is the node size, \( q' \) the cardinality of \( \mathcal{X}_\alpha \), \( N \) the global size and \( l \) the number of sampled rows used \( |\mathcal{T}_\alpha| \). [49]. We estimate the rank by \( \hat{\sigma}_s < \text{tol} \). Furthermore, we experimented by allowing larger blocks to have larger errors, i.e. \( \hat{\sigma}_s < \text{tol} \cdot \sqrt{\frac{N}{q'}} \).

We observed that in some test cases we could not match desired tolerance with required approximation rank, due to mis-estimation in that criterion. No final conclusion was drawn.

3.2.3. Target sampling

Sampling technique

To achieve reasonable computational effort we want to avoid a QR factorization using all corresponding rows; as a naive approximation one could subsample uniformly from indexes that do not belong to the node \( \alpha \). It has been reported that this scheme can be extremely inaccurate; nearest neighbor sampling is reported to be suitable for the geometric case [49] and ideally corresponds to leverage score sampling [46]. For each node a list of neighbors is calculated, from which we sample.

The intuition is that if the sample points accurately capture the row space of \( G \), then the skeletonization computed for the samples will accurately generalize to the entire matrix.

**Sampling neighbors for a leaf node:** We split each index nearest neighbor lists in pruning neighbors \( \mathcal{N}_p^i \) being the closer \( \kappa/2 \) indexes and \( \mathcal{N}_s^i \) being the further \( \kappa/2 \) neighbors. We first merge pruning neighbors for a leaf node to exclude from sampling, i.e.

\[
\mathcal{N}_p^\alpha = \bigcup_i \mathcal{N}_p^i \text{ where } i \in \alpha.
\]

Then we construct the sampling neighbors where we exclude the previously computed node pruning neighbors,

\[
\mathcal{N}_\alpha^\alpha = \{ \bigcup_i \mathcal{N}_s^i \} \setminus \mathcal{N}_p^\alpha \text{ where } i \in \alpha.
\]
3.2. Matrix compression

**Sampling neighbors for an interior node** We merge the sampling neighbor lists of the children and exclude pruning neighbors, i.e.

\[ N^s_\alpha = (N^s_{\text{left}} \cup N^s_{\text{right}}) \setminus (N^p_{\text{left}} \cup N^p_{\text{right}}). \]

We store interior pruning neighbors only for skeletons \( S_\alpha \), that are used in the parent node for \( \mathcal{X}_{\text{parent}} \). This also prevents pruning neighbor lists from becoming unnecessarily long. In particular,

\[ N^p_\alpha = \cup_i N^p_i \text{ where } i \in S_\alpha. \]

For leaves or interior points, if additional row samples are required, we choose them uniformly at random (excluding \( N^p_\alpha \) and \( i \in \alpha \)).

**Row adaptivity**

In a fixed row sampling case we usually sample \( l = 2q \) rows, where \( q = |S_\alpha| \). The following feature is not part of the C++ framework. We experimented with it using the Python reference code.

In some specific applications, we have a rough idea of necessary target samples, outgoing interactions or approximation rank. For a more general formulation with only few user-specified parameters, it is important to design an adaptive sampling mechanism. In [45] they stress the need for adaptive sampling mechanisms, but no implementation is given. In this section, we want to test whether chosen target samples accurately capture the row space of \( G \).

We check this by comparing \( \hat{G} = G_{\text{sampled}} \) and an updated factorization with appended samples \( \tilde{G} = G_{\text{addedSamples}} \).

\[
\begin{bmatrix}
\hat{G}^H \\
\tilde{G}
\end{bmatrix}
= [QR]
\begin{bmatrix}
\hat{G} \\
\tilde{G}
\end{bmatrix}
\]

We know that

\[
\begin{bmatrix}
\hat{G}^H \\
\tilde{G}
\end{bmatrix}
= [Q \ 0]
\begin{bmatrix}
R \\
\tilde{G}
\end{bmatrix}
\]

It remains to compute a QR of \( \begin{bmatrix} R \\ \tilde{G} \end{bmatrix} \) and look at the new triangular matrix \( \tilde{R} \).

We have experimented with this; ASKIT computes an estimated approximation rank using

\[ \tilde{s}_i = \tilde{R}_{ii} \left( \frac{q}{q'} \right)^{1/2} \left( \frac{N - q}{l'} \right)^{1/2} \]

where \( l' = l + |\text{addedSamples}| \). If the estimated rank \( \tilde{s} \) differs from the previously computed (by a tolerance \( c = 0, 1, 2, \ldots, 10 \)) we conclude that the chosen rows do not cover the row space sufficiently and we need more samples.

We have experimented with this but examples were found for which this strategy was not improving accuracy or not converging (many rows were added). Additionally, we have analyzed with a comparison of singular values. No final conclusion could be drawn.
3. Methods

3.3. Evaluation

In the evaluation phase we compute an approximate potential \( \tilde{u} = \tilde{K}w \) from a given vector of weights \( w \) (also often referred to as right hand side). For many applications it is required that we preserve symmetry.

3.3.1. Outgoing representation

In order to compute a low-rank evaluation we need to aggregate weights at skeletons. This is computed in a post-order traversal for the whole tree with the following relation

\[
\tilde{w}_{S(\alpha)} = P_{\alpha} \left[ \frac{\tilde{w}_{S(\text{left}(\alpha))}}{\tilde{w}_{S(\text{right}(\alpha))}} \right].
\]

3.3.2. Pruning rule

Classical neighbor-based pruning

Let’s introduce a list of leaves that are evaluated directly: \( \text{Near}(i) \) for an index \( i \) and \( \text{Near}(\alpha) \) for a leaf node \( \alpha \).

**Single evaluation:** We split the index-based all nearest neighbor lists in pruning \( \mathcal{N}_i^p \) of closer neighbors for pruning and \( \mathcal{N}_i^s \) for sampling. For the evaluation of a single point we can use a tree code implementation that prunes the tree according to \( \mathcal{N}_i^p \): every leaf that holds at least one index \( i \in \mathcal{N}_i^p \) is evaluated directly, the rest is considered far. Hence, \( \text{Near}(i) = \bigcup \text{LeafNode}(\mathcal{N}_i^p) \) where \( \text{LeafNode}(j) \) is a function that returns the leaf node of an index \( j \).

**Block evaluation:** In many cases, however, we want to evaluate in blocks: hence we build \( \text{Near}(\alpha) \) for each leaf node \( \alpha \) where we prune depending on leaf level nearest neighbor lists \( \mathcal{N}_\alpha^p \). Hence, we evaluate every leaf node \( \beta \) directly if it contains at least one entry from \( \mathcal{N}_\alpha^p \).

To enforce symmetry \( \text{Near}(.) \) lists have to be symmetric: If we evaluate node \( \beta \) directly from a tree node \( \alpha \), we also need to evaluate \( \alpha \) directly for \( \beta \).

For geometry-oblivious Gram vectors, we observed that the block evaluation pruning scheme grows extremely for bigger leaf sizes and larger number of neighbors. Hence, we introduced a budget.

**Budget pruning by frequency**

Notice that the size of \( \text{Near}(\beta) \) determines the number of off-diagonal direct evaluations. To prevent the cost from growing too fast, we introduce an user-defined parameter \textbf{budget} such that

\[
|\text{Near}(\beta)| < \text{budget} \cdot (N/m). \quad \text{(3.1)}
\]

While looping over \( i \in \mathcal{N}(\beta) \), instead of directly adding a \( \text{LeafNode}(i) \) to \( \text{Near}(\beta) \), we only mark it with a ballot. Then we insert candidates to \( \text{Near}(\beta) \) according to their votes until (3.1) is reached. To enforce symmetry of \( \tilde{K} \), we loop over all \( \text{Near} \) lists and enforce the following: if \( \alpha \in \text{Near}(\beta) \) then \( \beta \in \text{Near}(\alpha) \).
3.3. Evaluation

Far lists of nodes

We merge far lists in a postorder traversal. This means,

$$\text{Far}(\alpha) = \text{Far}(\text{left}) \cap \text{Far}(\text{right})$$

In the children’s far lists there remains

$$\text{Far}(\text{left}) = \text{Far}(\text{left}) \setminus \text{Far}(\alpha) \text{ and } \text{Far}(\text{right}) = \text{Far}(\text{right}) \setminus \text{Far}(\alpha)$$

3.3.3. Incoming representation

Before, we computed an interpolative decomposition of the matrix $\hat{G} = K(\mathcal{X}_\alpha, T_\alpha) \approx K(\mathcal{X}_\alpha, \mathcal{X}\setminus\mathcal{X}_\alpha)$ - the outgoing interaction of source block $\alpha$ towards distant target points. We build the incoming representation similarly by an interpolative decomposition, since $K$ is symmetric we can write

$$G \approx P^T G_{\text{row}}$$

**Skeleton to skeleton:** We compute for targets of this node a skeleton potential of a node $\beta$. We use $\bar{u}$ to denote skeleton values, i.e.

$$\bar{u}_\alpha = \sum_{\beta \in \text{Far}(\alpha)} K_{\alpha,\beta} \bar{w}_\beta$$

**Skeleton to nodes:** Next, we use the incoming representation to approximate the potentials at the children. Starting from root, we perform a preorder traversal, where we accumulate

$$[\bar{u}_{\text{left}}, \bar{u}_{\text{right}}] + = P^T_{\beta} \bar{u}_\beta$$

On the leaf level, this directly accumulates the output result, i.e. $u_\alpha = P^T_{\alpha} \bar{u}_\alpha$.

The three tasks (outgoing representation & skeleton to skeleton & skeleton to nodes) compute all matvec operations for the far-field nodes. All matvec on $K_{\beta,\alpha}$ in $\text{Near}(\beta)$ are computed by Leaf To Leaf operation and directly accumulated on leaves to $u_\beta$.

Telescoping relations

Here we describe an alternative relation that gives us more flexibility than a standard tree traversal. As a consequence of the nesting property, $S_\alpha \subset S_{\text{left}} \cup S_{\text{right}}$ we can construct a telescoping expression for the full coefficient matrix.

We never explicitly form $P_{S_\alpha,\alpha}$, but instead we can use the telescoping expression during evaluation.

We start by introducing the notation $\hat{G} = G_{T_\alpha, [S_{\text{left}}, S_{\text{right}}]}$ where $[S_{\text{left}}, S_{\text{right}}]$ contains the skeletons of the children of $\alpha$ and $T_\alpha$ the target samples. The applied interpolative decomposition can be written as

$$G_{T_\alpha, [S_{\text{left}}, S_{\text{right}}]} \approx G_{T_\alpha, S_\alpha} P_{S_\alpha, [S_{\text{left}}, S_{\text{right}}]}$$

where $P_{S_\alpha, [S_{\text{left}}, S_{\text{right}}]}$ is the projection matrix relating $\mathcal{X}_\alpha = S_{\text{left}} \cup S_{\text{right}}$ to $S_\alpha$. We follow this line and construct an approximation of the full block $G_{T_\alpha, \mathcal{X}_\alpha}$:

$$G_{T_\alpha, \mathcal{X}_\alpha} \approx G_{T_\alpha, [S_{\text{left}}, S_{\text{right}}]} \begin{bmatrix} P_{S_{\text{left}}, \text{left}} & \vdots & \vdots & \vdots & \vdots \\ \vdots & P_{S_{\text{left}}, \text{right}} & \vdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & P_{S_{\text{right}}, \text{left}} & \vdots \\ \vdots & \vdots & \vdots & \vdots & P_{S_{\text{right}}, \text{right}} \end{bmatrix}$$
3. Methods

Consecutively applied, this leads to a *telescoping* expression for the full coefficient matrix:

\[
P_{S_n\alpha} = P_{S_n[S_{e,l},S_{e,r}]} \begin{bmatrix} P_{S_{e,l}\text{left}} & P_{S_{e,r}\text{right}} \end{bmatrix}
\]

**Efficient implementation**

The evaluation step is hence split in four phases, sketched in algorithm 3; it follows the fast multipole scheme from [49].

**Algorithm 3 Evaluate \((u,w)\)**

1. (POST) \(N2S(\alpha)\) \(\triangleright\) compute skeleton weights \(\tilde{w}\)
2. (ANY) \(S2S(\beta)\) \(\triangleright\) apply skeleton basis \(K_{\tilde{\beta}\tilde{\alpha}}\)
3. (PRE) \(S2N(\beta)\) \(\triangleright\) accumulate skeleton potentials \(\tilde{u}\)
4. (ANY) \(L2L(\beta)\) \(\triangleright\) accumulate direct \texttt{matvec} to \(u\)

The first step is to perform a postorder traversal \(N2S(\alpha)\) (Nodes To Skeletons) that we described as *outgoing representation*. We compute skeleton weights \(\tilde{w}_\alpha = P_{S_n,\alpha}w_\alpha\) or in a postorder fashion from its children \(\tilde{w}_\alpha = P_{\alpha,[S_{e,l},S_{e,r}]}[\tilde{w}_{\text{left}},\tilde{w}_{\text{right}}]\) for each tree node.

\(S2S(\beta)\) (Skeletons to Skeletons) applies the skeleton basis \(K_{\tilde{\beta}\tilde{\alpha}}\) and accumulates skeleton potentials \(\tilde{u}\) for each node: \(\tilde{u}_\beta = \sum_{\alpha \in \text{Far}(\beta)} K_{\tilde{\beta}\tilde{\alpha}} \tilde{w}_\alpha\). We can compute it in any order after \(\tilde{w}_\alpha\) are computed.

\(S2N(\beta)\) (Skeletons To Nodes) performs interpolation on the left and accumulate \(\tilde{u}\) with a preorder traversal. For each node \(\beta\), we accumulate \([\tilde{u}_{\text{left}},\tilde{u}_{\text{right}}] + P_{S_n[S_{e,l},S_{e,r}]}^T \tilde{u}_\beta\) to its children.

In the leaf node this directly accumulates to the output result.

For an efficient implementation we need accumulated lists (block evaluation vs. single point evaluation). Furthermore, instead of traversing in a tree we address nodes using their MortonID.

3.4. Parallel schemes

A tree traversal may exhibit high parallelism at the leaf level but due to the dependencies the parallelism typically diminishes near the root level. 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. We observed significant workload variations during the compression (due to adaptive approximation ranks) and during the evaluation (tasks \(N2S\) and \(S2N\)).

One solution for diminishing parallelism towards the root level is to exploit parallelism in finer granularity. For example, when the number of tree nodes in the single 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 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 task scheduling. To this end we test
3.4. Parallel schemes

We split the algorithm in Compression and Evaluation. For compression we only use the BLAS/LAPACK routines: GEQP3 for a pivoted QR and triangular solve TRSM. The pivoted QR is executed in a postorder traversal, TRSM in any order, after node’s skeletons (GEQP3) are computed.

After the compression we perform an evaluation – we listed dependency relations in algorithm 3. We identify three different phases: node to skeletons (N2S), skeletons to skeletons (S2S) and skeletons to nodes (S2N).

A dependency graph for the evaluation phase is listed in Figure 3.2. 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 for Compression and Evaluation. For simplicity, below we only discuss the evaluation phase for the HSS case (neighbor-based pruning leads to more complicated graphs).

Figure 3.2 depicts task dependencies during the evaluation phase for N2S, S2S and S2N where the off-diagonal blocks are low-rank (HSS) with $S = 0$. A task dependency tree is generated by our runtime using symbolic traversals. The N2S, S2S, and S2N execution order is performed on a binary tree.

We use three symbolic traversals of algorithm 3. In the first traversal (postorder) we find

---

1This work was collectively developed and mainly carried out by Chenhan D. Yu and is mainly addressed in a SC’17 paper [17]. We only cover the theoretical aspects here.

2Execution order from left to right: dependencies are easier to follow if one rotates the page by 90° counterclockwise.
3. Methods

that \( \tilde{w}_{\text{left}} \) is written by \( \text{left} \). Going from \( \tilde{w}_{\text{left}} \) to \( \tilde{w}_\beta \), we observe that \( \tilde{w}_{\text{left}} \) is read by \( \beta \), i.e. \( \tilde{w}_\beta = P_{\beta[\text{left} \oplus \text{right}]}[\tilde{w}_{\text{left}}; \tilde{w}_{\text{right}}] \). This RAW dependency is shown in the DAG with the edge from \( \text{left} \) to \( \beta \).

Intertask dependencies are discovered by the symbolic execution of the yellow tree. At node \( \beta \) (in yellow), the relation \( \tilde{u}_\alpha = K_{\alpha \beta \tilde{w}_\beta} \) will read \( \tilde{w}_\beta \). Again this is a RAW dependency, hence the edge from the blue \( \beta \) to the yellow \( \alpha \). The whole dependency graph for steps 1–3 is built after the green postorder tree traversal. Step 4 in algorithm 3 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.

**Runtime.** 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) [63] using OpenMP threads. Each worker (thread) in the runtime 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 DAG will be dispatched to a “ready” queue. Each worker keeps consuming tasks in its own queue until no tasks are left.

The runtime of 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 DAG is assigned to a 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.

3.5. Experimental setup

Results shown in this thesis were conducted on TACC’s Lonestar 5, (two 12-core, 2.6GHz, Xeon E5-2690 v3 “Haswell” 64 GB DDR4-2133 – 8 x 8GB dual rank x8 DIMMS). Experiments were conducted on the following 16 matrices, summarized in Table 3.1. Additionally, we examined dense graph problems (Laplacian of a graph, denoted \( G_{01}, G_{02}, \ldots \)). The matrices are normed to unity.

Throughout this thesis we use a relative error \( \epsilon_2 \) which we compute by comparing to a sampled exact \text{matvec} evaluation. Since we offer multiple right hand sides (denoted as number of right hand sides := \( n_{\text{Rhs}} \)), this can either be viewed as consecutive \text{matvec} evaluation or a matrix-matrix product. Hence, we define \( \epsilon_2 \) as

\[
\epsilon_2 := \frac{\| \tilde{K} w - K w \|_F}{\| K w \|_F}, \quad \text{where } w \in \mathbb{R}^{N \times n_{\text{Rhs}}}
\]

This metric requires \( O(r N^2) \) work; to reduce the computational effort we instead sample 100 rows of \( K \).

The adaptive tolerance \( \tau \), reflects the error of the subsampled block and may not correspond to the output error \( \epsilon_2 \). Depending on the problem, \( \tau \) may misestimate the rank.
3.5. Experimental setup

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01</td>
<td>Sparse forward 2D Poisson problem operator</td>
</tr>
<tr>
<td>K02</td>
<td>2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid.</td>
</tr>
<tr>
<td>K03</td>
<td>Same setup with the oscillatory Helmholtz operator and 10 points per wave length</td>
</tr>
<tr>
<td>K04-K06</td>
<td>Kernel matrices in six dimensions: Gaussians with different bandwidths, narrow and wide</td>
</tr>
<tr>
<td>K07</td>
<td>Kernel matrices in six dimensions: Laplacian Green’s function</td>
</tr>
<tr>
<td>K08-K10</td>
<td>Kernel matrices in six dimensions: Quadratic, inverse quadratic, and polynomial kernel</td>
</tr>
<tr>
<td>K11</td>
<td>Inverse squared 1D variable coefficient Poisson problem operator</td>
</tr>
<tr>
<td>K12-14</td>
<td>2D advection-diffusion operators on a regular grid with highly variable coefficients</td>
</tr>
<tr>
<td>K15 &amp; 16</td>
<td>2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients</td>
</tr>
</tbody>
</table>

Table 3.1.: Test matrices that are created with a MATLAB script and stored binary in floating point numbers. In order to reduce file sizes we mostly used single precision format.
3. Methods
4. Numerical Experiments

Next, we present the numerical results generated in the study. We first discuss our geometry-oblivious approach using the Gram vector idea. We then continue with evaluation accuracy and performance results, generated with the C++ framework that has been collectively developed with Chenhan Yu and James Levitt. Last, we compare the results to the existing software HODLR [4] and STRUMPACK [60].

4.1. Evaluation of Gram vector approach

Using the Gram vector space for a FMM is unique. First, since we allow user input, we can assume that the numbering of matrices may not be optimal. In some cases the user may specify a beneficial ordering, but here we show that through Gram vector partitioning we can recover blocks that allow low-rank approximation.

Second, we observe that target sampling from Gram vectors improves accuracy of the overall scheme. We show that for a HSS-case only. In the paper [17] we have shown, that nearest-neighbor pruning gives significant benefit compared to an HSS approach.

4.1.1. Hierarchical pruning

In Figure 4.1 we have plotted the spectra of singular values of a first level off-diagonal block. We want to approximate this block with a low-rank approximation, so it would be beneficial that we can approximate this block with a rank \( r << \frac{N}{2} \).

This corresponds to the maximum achievable compression ratio. Note that here we compare to a singular value decomposition. Using the interpolative decomposition we may need higher approximation ranks as we use columns of \( G \) as a basis. That means \( \| G \text{col}_P - G \|_2 \leq \sigma_{s+1} \sqrt{s(n - s) + 1} \) where \( s \) is the approximation rank and \( n \) the size of the approximated matrix. Hence, the interpolative decomposition may need larger approximation ranks, bounded by the factor \( \sqrt{s(n - s) + 1} \).

Coming back to Figure 4.1, we can see that the spectra of the singular values are different for each test case (we only show the interesting cases here). Note that as projection axis we chose \( \phi_p - \phi_k \), where \( \phi_p \) is furthest away from an approximate center of mass (and \( k \) at opposite end). In case (a) we observe that Gram angles partitioning performs similarly well for a low-rank approximation as a lexicographic ordering for a regular 2D grid – expectedly this is (almost) optimal on the first level. A Gram \( \ell^2 \) partitioning here does worse than a random ordering.

In (b) we can see that all schemes are (almost) similar. Gram \( \ell^2 \) is actually nearly equivalent to a geometric split and Gram angles is slightly worse. For 10D random points (from a Gaussian normal distribution) a first split is not very meaningful. But we see the trend of good performance of Gram \( \ell^2 \). We see a similar trend in (c). For (d) we again have
4. Numerical Experiments

a regular 2D grid. We see good partitioning behavior on the first level for lexicographic (expectedly), and Gram $ℓ^2$ and Gram angles are notably better than a random split. In general, we found it hard to draw conclusions, and questioned if this approach of analyzing only the first level was too simplistic.

We hoped to gain a better insight to the Gram idea by applying it using the full ASKIT inspired approach.

4.1.2. Sampling Strategy

Sampling from geometric neighbors has been used in ASKIT and is shown to be improving accuracy of the scheme significantly [48, 51]. In this section we want to show that sampling from Gram vector space defined neighbors can be beneficial as well. In Table 4.1 we have used the Gram angle criteria for neighbors, and report accuracies for an HSS case. We expect that neighbor-based sampling shows improvements from $k = 0$ to $k = 4$. Since we use a fixed sampling size of $l = 2q$ we need at least $2m$ node neighbors in leaves. For $k = 4$ we hope that we satisfy this, and do not sample at random from uniform (most naive approach). We expect not necessarily an improvement for larger $k$, but error should not increase significantly (approximation error subject to randomness).

We can see that, for some cases, as we increase neighbors our approximation accuracy gets better. However, the results vary significantly. We suspect that the greatest contribution to the error originates from the first level 50:50 split; we plan that in the future we would like to steer pruning neighbors $κ_{pr}$ and $κ_{sa}$ separately.

For target sampling, we remained with open research ideas for future work. Gram neighbors – be it Gram $ℓ^2$ or Gram angle – are different from the nature of geometric neighbors. In order to analyze their significance for sampling one could compare the samples to leverage scores. Furthermore, we experimented with the idea of a double pass where we sample from skeletons of a previous (inaccurate) run.

4.2. Performance Results

With preliminary ideas about the Gram vector space, we straightforwardly implemented the algorithm and applied it to a variety of physical cases. In the following we (4.2.1) show that our algorithm can mostly satisfy desired error bounds and (4.2.2) outperforms existing software.

4.2.1. Evaluation Accuracy

In Table 4.2 we applied the Gram $ℓ^2$ criterion, and in Table 4.3 the angle-based approach. We can see differences for some cases, but it turns out that they are very similar. A very un-interesting case is K01, because it is a sparse matrix (2D Poisson). For many other matrices (K02-K12) we see that the error almost satisfies a specified approximation tolerance $τ = 1E − 6$. Especially, it also works for the Laplacian kernel (K07) which is hard to compress. We can see that for K13-16 we cannot satisfy error bounds. We observe that K15 and K16 (2D pseudo-spectral advection-diffusion-reaction operator) Gram angles perform
4.2. Performance Results

Figure 4.1: $N = 1024$. Singular values $\sigma_i$ (normalized to $\sigma_1$) are plotted in double logarithmic scale for a first level off-diagonal block ($\frac{N}{2} \times \frac{N}{2}$).
4. Numerical Experiments

<table>
<thead>
<tr>
<th>case</th>
<th>k</th>
<th>$\varepsilon_2$</th>
<th>k</th>
<th>$\varepsilon_2$</th>
<th>k</th>
<th>$\varepsilon_2$</th>
<th>k</th>
<th>$\varepsilon_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01N65536</td>
<td>0</td>
<td>8E−1</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>K02N65536</td>
<td>0</td>
<td>2E−4</td>
<td>4</td>
<td>6E−4</td>
<td>8</td>
<td>2E−4</td>
<td>16</td>
<td>8E−5</td>
</tr>
<tr>
<td>K03N65536</td>
<td>0</td>
<td>4E−10</td>
<td>4</td>
<td>1E−6</td>
<td>8</td>
<td>1E−6</td>
<td>16</td>
<td>1E−6</td>
</tr>
<tr>
<td>K04N65536</td>
<td>0</td>
<td>3E−5</td>
<td>4</td>
<td>4E−5</td>
<td>8</td>
<td>6E−5</td>
<td>16</td>
<td>1E−4</td>
</tr>
<tr>
<td>K05N65536</td>
<td>0</td>
<td>5E−5</td>
<td>4</td>
<td>8E−5</td>
<td>8</td>
<td>9E−5</td>
<td>16</td>
<td>1E−4</td>
</tr>
<tr>
<td>K06N65536</td>
<td>0</td>
<td>1E−1</td>
<td>4</td>
<td>9E−2</td>
<td>8</td>
<td>5E−2</td>
<td>16</td>
<td>3E−2</td>
</tr>
<tr>
<td>K07N65536</td>
<td>0</td>
<td>2E−5</td>
<td>4</td>
<td>2E−5</td>
<td>8</td>
<td>9E−5</td>
<td>16</td>
<td>1E−4</td>
</tr>
<tr>
<td>K08N65536</td>
<td>0</td>
<td>3E−5</td>
<td>4</td>
<td>3E−5</td>
<td>8</td>
<td>4E−5</td>
<td>16</td>
<td>4E−5</td>
</tr>
<tr>
<td>K09N65536</td>
<td>0</td>
<td>3E−5</td>
<td>4</td>
<td>5E−5</td>
<td>8</td>
<td>5E−5</td>
<td>16</td>
<td>4E−5</td>
</tr>
<tr>
<td>K10N65536</td>
<td>0</td>
<td>10E−7</td>
<td>4</td>
<td>8E−7</td>
<td>8</td>
<td>9E−7</td>
<td>16</td>
<td>10E−7</td>
</tr>
<tr>
<td>K11N65536</td>
<td>0</td>
<td>1E−4</td>
<td>4</td>
<td>1E−4</td>
<td>8</td>
<td>3E−4</td>
<td>16</td>
<td>5E−4</td>
</tr>
<tr>
<td>K12N65536</td>
<td>0</td>
<td>1E−4</td>
<td>4</td>
<td>7E−5</td>
<td>8</td>
<td>2E−4</td>
<td>16</td>
<td>4E−4</td>
</tr>
<tr>
<td>K13N65536</td>
<td>0</td>
<td>1E−1</td>
<td>4</td>
<td>2E−1</td>
<td>8</td>
<td>3E−1</td>
<td>16</td>
<td>1E−1</td>
</tr>
<tr>
<td>K14N65536</td>
<td>0</td>
<td>3E−1</td>
<td>4</td>
<td>2E−1</td>
<td>8</td>
<td>9E−2</td>
<td>16</td>
<td>2E−1</td>
</tr>
<tr>
<td>K15N65536</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>4E−1</td>
<td>16</td>
<td>2E−1</td>
</tr>
<tr>
<td>K16N65536</td>
<td>0</td>
<td>6E−1</td>
<td>4</td>
<td>1E1</td>
<td>8</td>
<td>6</td>
<td>16</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.1.: HSS (Budget = 0). We use a neighbor-based sampling strategy. $k$ denotes the number of neighbors and here we use all neighbors for sampling (no 50:50 split for pruning neighbors). If we do not have enough sampling neighbors we fill up the samples using uniform sampling – in $k = 0$ we use a complete uniform sampling. The size of the matrix is $N \times N$ where $N = 65536 \approx 65k$. 

36
4.2. Performance Results

Table 4.2.: Performance results for $N = 16384$ matrices using $tol = 1E-6 < R_{ii,m} = 512$ and $nRhs = 512$. Here we use Gram $l^2$ neighbors and partitioning. We allow a budget of direct evaluation of 10%. $\%K$ is the amount of direct evaluation which may be twice the budget due to symmetrizing Near-Far lists.

<table>
<thead>
<tr>
<th>case</th>
<th>$k$</th>
<th>$\epsilon_2$</th>
<th>$%K$</th>
<th>$t_{ANN}$</th>
<th>$t_{setup}$</th>
<th>$t_{skel}$</th>
<th>$t_{eval}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01N16384</td>
<td>16</td>
<td>1</td>
<td>18.1%</td>
<td>0.7</td>
<td>0.4</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K02N16384</td>
<td>16</td>
<td>3E-6</td>
<td>1.7%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.8</td>
<td>0.0</td>
</tr>
<tr>
<td>K03N16384</td>
<td>16</td>
<td>5E-8</td>
<td>17.7%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K04N16384</td>
<td>16</td>
<td>2E-6</td>
<td>12.9%</td>
<td>0.5</td>
<td>0.3</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>K05N16384</td>
<td>16</td>
<td>8E-6</td>
<td>12.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K06N16384</td>
<td>16</td>
<td>2E-2</td>
<td>13.6%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.2</td>
<td>0.0</td>
</tr>
<tr>
<td>K07N16384</td>
<td>16</td>
<td>4E-4</td>
<td>19.0%</td>
<td>0.5</td>
<td>0.3</td>
<td>1.4</td>
<td>0.0</td>
</tr>
<tr>
<td>K08N16384</td>
<td>16</td>
<td>3E-6</td>
<td>18.7%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K09N16384</td>
<td>16</td>
<td>2E-5</td>
<td>12.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K10N16384</td>
<td>16</td>
<td>8E-7</td>
<td>18.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.8</td>
<td>0.0</td>
</tr>
<tr>
<td>K12N16384</td>
<td>16</td>
<td>1E-5</td>
<td>1.5%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K13N16384</td>
<td>16</td>
<td>2</td>
<td>4.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>K14N16384</td>
<td>16</td>
<td>2</td>
<td>4.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>K15N16384</td>
<td>16</td>
<td>2</td>
<td>19.1%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>K16N16384</td>
<td>16</td>
<td>1</td>
<td>19.0%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.4</td>
<td>0.0</td>
</tr>
</tbody>
</table>

significantly better than Gram $l^2$, but still cannot satisfy error bounds. In K13 and K14 we underestimate the approximation rank significantly. If one changes the adaptivity criterion in a very strict way, or if we perform a fixed-rank approximation, we can reduce the error significantly.

4.2.2. Comparison to other software

In the following we compare our algorithm to existing software packages. We start by explaining the codes and show results, and close the section with a performance comparison. Both software packages mainly address hierarchical solvers (using the Sherman-Morrison-Woodbury formula), but since we do not support this, we neglect this here. We only look at the matvec operation (or matmat since both codes offer multiple right hand sides). These codes were adapted such that they run on any input matrix from memory.

HODLR and STRUMPACK resemble the classical HSS idea on the given ordering, without any neighbor-based sampling or pruning. In the paper, we have shown that for some of these cases (e.g. K02 and K15) using neighbor-based pruning (i.e. direct evaluation) is highly beneficial. The only tunable “knob” these codes can offer is to increase the approximation rank.
4. Numerical Experiments

<table>
<thead>
<tr>
<th>case</th>
<th>k</th>
<th>$\epsilon_2$</th>
<th>%K</th>
<th>$t_{\text{ANN}}$</th>
<th>$t_{\text{setup}}$</th>
<th>$t_{\text{skel}}$</th>
<th>$t_{\text{eval}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01N16384</td>
<td>16</td>
<td>8E–1</td>
<td>17.6%</td>
<td>0.7</td>
<td>0.4</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K02N16384</td>
<td>16</td>
<td>5E–5</td>
<td>11.3%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.7</td>
<td>0.0</td>
</tr>
<tr>
<td>K03N16384</td>
<td>16</td>
<td>4E–8</td>
<td>16.6%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K04N16384</td>
<td>16</td>
<td>2E–6</td>
<td>14.2%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>K05N16384</td>
<td>16</td>
<td>5E–6</td>
<td>13.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K06N16384</td>
<td>16</td>
<td>2E–2</td>
<td>13.8%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K07N16384</td>
<td>16</td>
<td>2E–4</td>
<td>15.8%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K08N16384</td>
<td>16</td>
<td>3E–6</td>
<td>15.3%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.2</td>
<td>0.0</td>
</tr>
<tr>
<td>K09N16384</td>
<td>16</td>
<td>2E–5</td>
<td>13.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>K10N16384</td>
<td>16</td>
<td>8E–7</td>
<td>15.5%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>K12N16384</td>
<td>16</td>
<td>1E–5</td>
<td>7.7%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K13N16384</td>
<td>16</td>
<td>1</td>
<td>17.9%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K14N16384</td>
<td>16</td>
<td>1</td>
<td>17.8%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K15N16384</td>
<td>16</td>
<td>6E–2</td>
<td>11.3%</td>
<td>0.4</td>
<td>0.3</td>
<td>1.1</td>
<td>0.0</td>
</tr>
<tr>
<td>K16N16384</td>
<td>16</td>
<td>4E–1</td>
<td>14.1%</td>
<td>0.4</td>
<td>0.3</td>
<td>0.8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 4.3: Performance results for $N = 16384$ matrices using $\text{tol} = 1E - 6 < R_{ii,m} = 512$ and $nRhs = 512$. Here we use Gram angles for neighbors and partitioning. We allow a budget of direct evaluation of 10%. %K is the amount of direct evaluation which may be twice the budget due to symmetrizing Near-Far lists.

HODLR

This code can work with any compression technique. In the paper [5] they stress an analytic compression using Chebychev polynomials. In the code repository they offer algebraic low-rank approximations of the appropriate blocks using a partial pivoted LU algorithm (a.k.a. adaptive cross approximation) and the Eigen library.

It compresses $A \approx \tilde{A}_r = (LU)_r$ of rank $r$, where $L$ is lower left triangular (with $r$ columns) and $U$ upper right triangular (with $r$ rows). The computed rank is determined from a stopping criterion. We explain that in the following where we use the notation $A_r$ which denotes the permuted $r \times r$ submatrix of $A$.

A pseudo-code for the partial pivoted LU algorithm is given in algorithm 4. As for notation we use $\| (\cdot)_{r \cdot} \| / \| (\cdot)_{r :} \|$ as column/row norm of the current “update” and $A_{rr}$ as the pivot.

**Algorithm 4** partial\_pivoted\_LU

1: do
2: select pivot element (max entry in respective column)
3: calculate next column $r$ in $L$ and next row $r$ in $U$
4: while $\frac{\| (A_r - A_r^\ast)_{\cdot r} \|}{\| (A_r - A_r^\ast)_{r \cdot} \|} \geq A_{rr} \ast \text{tol}$

By construction of this algorithm, the necessary operations are very problem-dependent. No sampling takes place, so it may happen that approximation ranks are the size of the
4.2. Performance Results

<table>
<thead>
<tr>
<th></th>
<th>MatVec $e_2$</th>
<th>$\tau = 1E - 6$</th>
<th>time compression</th>
<th>$t_{\text{fastMatVec}}$</th>
<th>$t_{\text{exactMatVec}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01N16384</td>
<td>2E–5</td>
<td>1</td>
<td>0.05</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>K02N16384</td>
<td>6E–1</td>
<td>0.39</td>
<td>0.05</td>
<td>0.23</td>
<td></td>
</tr>
<tr>
<td>K03N16384</td>
<td>2E–16</td>
<td>0.4</td>
<td>0.04</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K04N16384</td>
<td>9E–2</td>
<td>0.58</td>
<td>0.05</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>K05N16384</td>
<td>6E–2</td>
<td>1.59</td>
<td>0.09</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K06N16384</td>
<td>3E–4</td>
<td>1131.16</td>
<td>1.92</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>K07N16384</td>
<td>9E–1</td>
<td>0.37</td>
<td>0.04</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K08N16384</td>
<td>1E–1</td>
<td>0.42</td>
<td>0.04</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K09N16384</td>
<td>2E–2</td>
<td>11.26</td>
<td>0.21</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K10N16384</td>
<td>6E–1</td>
<td>0.55</td>
<td>0.05</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K11N16384</td>
<td>7E–6</td>
<td>0.4</td>
<td>0.04</td>
<td>0.23</td>
<td></td>
</tr>
<tr>
<td>K12N16384</td>
<td>3E–1</td>
<td>0.4</td>
<td>0.05</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>K13N16384</td>
<td>4E–6</td>
<td>0.41</td>
<td>0.04</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>K14N16384</td>
<td>3E–6</td>
<td>0.43</td>
<td>0.05</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>K15N16384</td>
<td>2E–1</td>
<td>2.37</td>
<td>0.12</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>K16N16384</td>
<td>4E–1</td>
<td>0.41</td>
<td>0.05</td>
<td>0.23</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: HODLR results. $\tau = 1E - 6$, $m = 1024$, $nRhs = 1$. Time metrics in seconds. Applied to matrices with initial (lexicographic) ordering.

full block. We applied this code to all test cases for $N = 16384$. Results are tabulated in Table 4.4. We can see that compression and evaluation times vary significantly. Also, the specified tolerance does not correspond to the matvec error. For cases where the error is high, usually the approximation rank is very small – not enough to reconstruct the matrix-vector product accurately. We also observe that for most cases the fast matrix vector product is significantly faster than the regular multiplication.

**STRUMPACK**

STRUMPACK is a distributed MPI code that uses ScALAPACK for distributed linear algebra operations. The memory splitting onto processors is done using BLACS. For compression of HSS blocks it uses a randomized interpolative decomposition (multiplies with randomized vector $R_{r|c}$) that is described here [55]. We give a pseudo-code of the algorithm in 5.

The approximation rank is determined by $\frac{R_{r}}{R_{i}} < tol$. The code offers adaptive sampling: When the rank at a given node $r_{fail}$ is too large, add new columns to $R_{r}$ and $R_{c}$, compute the new columns of $S_{r}$ and $S_{c}$ with a product, and restart the postorder traversal. However, it has been found that adaptive sampling takes much time and it starts to aggregate memory. For a few cases this lead to memory allocation errors. To avoid this, we can set the minimum random sample size higher, and the adaptivity to adding more samples per iteration (more aggressive). This solution is not optimal but it worked for our cases.

In Table 4.5 we have plotted the results for STRUMPACK of the 16 mentioned cases. However, K03 is exempted because STRUMPACK returns a bug (sgemms are called for illegal
4. Numerical Experiments

Algorithm 5 randomized_interpolative_decomposition
1: Generate $R_r$ and $R_c$ random $n \times d$ matrices.
2: Compute the samples $S_r = AR_r$ and $S_c = AR_c$
3: Traverse the tree in topological order (i.e., children before parents): at each node
   a. Construct local samples.
   b. Compute generators using Interpolative Decomposition.
   c. Update samples and random vectors to make the construction of local samples faster at subsequent nodes

<table>
<thead>
<tr>
<th></th>
<th>MatVec $\epsilon_2$</th>
<th>time$_{compression}$</th>
<th>time$_{exactMatVec}$</th>
<th>time$_{fastMatVec}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>K01N16384</td>
<td>6E−8</td>
<td>2.3</td>
<td>2E−2</td>
<td>6E−3</td>
</tr>
<tr>
<td>K02N16384</td>
<td>7E−5</td>
<td>1.1</td>
<td>2E−2</td>
<td>6E−3</td>
</tr>
<tr>
<td>K04N16384</td>
<td>3E−4</td>
<td>1.8</td>
<td>2E−2</td>
<td>8E−3</td>
</tr>
<tr>
<td>K05N16384</td>
<td>2E−3</td>
<td>3.6</td>
<td>2E−2</td>
<td>10E−3</td>
</tr>
<tr>
<td>K06N16384</td>
<td>3E1</td>
<td>137.7</td>
<td>2E−2</td>
<td>3E−2</td>
</tr>
<tr>
<td>K07N16384</td>
<td>5E−2</td>
<td>17.7</td>
<td>7E−2</td>
<td>2E−2</td>
</tr>
<tr>
<td>K08N16384</td>
<td>2E−3</td>
<td>10.8</td>
<td>7E−2</td>
<td>2E−2</td>
</tr>
<tr>
<td>K09N16384</td>
<td>8E−3</td>
<td>13.6</td>
<td>7E−2</td>
<td>2E−2</td>
</tr>
<tr>
<td>K10N16384</td>
<td>1E−4</td>
<td>2.0</td>
<td>4E−2</td>
<td>7E−3</td>
</tr>
<tr>
<td>K11N16384</td>
<td>2E−6</td>
<td>1.9</td>
<td>3E−2</td>
<td>1E−2</td>
</tr>
<tr>
<td>K12N16384</td>
<td>8E−5</td>
<td>2.0</td>
<td>3E−2</td>
<td>7E−3</td>
</tr>
<tr>
<td>K13N16384</td>
<td>8E−7</td>
<td>1.9</td>
<td>3E−2</td>
<td>7E−3</td>
</tr>
<tr>
<td>K14N16384</td>
<td>9E−7</td>
<td>1.9</td>
<td>3E−2</td>
<td>1E−2</td>
</tr>
<tr>
<td>K15N16384</td>
<td>4E−4</td>
<td>33.0</td>
<td>2E−2</td>
<td>2E−2</td>
</tr>
<tr>
<td>K16N16384</td>
<td>2E−4</td>
<td>33.5</td>
<td>4E−2</td>
<td>2E−2</td>
</tr>
</tbody>
</table>

Table 4.5: STRUMPACK results. $\tau = 1E−6$. $m = 1024$, nRhs = 1. Initial ordering; tol = 1e−6; m = 1024. Applied to matrices with initial (lexicographic) ordering.

parameters due to approximation rank 0). We can see that this HSS approach (uniform sampling) has particular problems for the kernel cases (K04-K10); the codes are applied to the initial ordering (which is random). Additionally, it seems that the approach of multiplication with a uniform sampled vector is not working very well. Neighbor information is necessary. However, we also see that STRUMPACK performs very well for K13-16 – the cases where GOFMM did not perform very well. HODLR performed reasonably well for K13-14, but failed for K15-16 (see Table 4.4). STRUMPACK also performs well for cases K15-16 but the compression time (and the matvec multiplication time) is comparably large. Note that fast matrix vector product pay off compared to exact (PBLAS) for multiple right hand sides.
4.2. Performance Results

Table 4.6.: Wall-clock time comparison of the three software packages for selected cases.

For K02-K12, we use $N = 36K$, K17 $N = 32K$, and G03 $N = 65K$. For all software, we use leaf node size $m = 512$ and $nRhs = 1024$. We control internal parameters for each software to target the same relative error ($1E^{-4}$). For GOFMM we use 16 neighbors and the Gram angle criterion.

**Direct comparison**

Next, for selected cases we compare the performance to our algorithm GOFMM. Performance results are tabulated in Table 4.6. We attempted to fix the desired error bound – and iteratively adjust parameters such that we satisfy these bounds. For HODLR we only steered the error by the approximation tolerance $\tau$; for STRUMPACK we could not satisfy error bounds only by the approximation tolerance $\tau$. For cases that show exorbitant compression times we had to increase the sampling size to almost complete block size. Note that for compression time of GOFMM a significant amount of time (> 50%) is consumed by the all nearest neighbor computations – a section of the code which we did not optimize.

In Table 4.6 we observe that HODLR’s compression is very competitive – only for K17 (a pseudo-spectral operator) the adaptive cross approximation approach does not work very well. Evaluation times for HODLR are significantly larger than GOFMM and (for some) STRUMPACK. It seems that they have not optimized this part. STRUMPACK’s compression times are for some cases extremely high – we could not satisfy error bounds and increased sampling sizes. Most of the time is consumed by a matrix-matrix product of the random sample matrices with the test matrix. The code only requires a routine to do that function, and report significant benefits for cases where they have a fast routine of this step. Nevertheless, for some cases STRUMPACK could not satisfy error bounds.

Lastly, regarding GOFMM: In the compression phase our approach is competitively fast. In the evaluation phase (which we particularly optimized) our code’s performance is significantly faster than the two other software packages.
4. Numerical Experiments
5. Conclusion and future work

5.1. Conclusion

Fast multipole methods (FMM) are applicable in many domains – simulations of gravity and Coulomb potentials, waves and scattering, or fluids and transport, or in data analysis for machine learning, geostatistics, and image analysis. Various code frameworks exist for specific problems [16, 21]. Algebraic approaches generalize approximation efforts – and pave the way for kernel-independent methods [66, 24, 51].

Nevertheless, for most code frameworks problem-specific knowledge and geometrical information is required – it can be cumbersome to integrate them into existing research codes. The idea of this project was to design a “black-box” algorithm, where no distances in the original space will be needed. We only require a routine that provides access to matrix entries. We compare ourselves to the class of hierarchical matrices [26, 5], but allow a more elaborate Near-far pruning criterion.

We extended research efforts from ASKIT – a high dimensional approach which uses nearest-neighbor pruning [51]. Instead of geometric knowledge of the point distribution, we employ the fact that every SPD matrix can be viewed as a Gramian matrix – where entries $K_{ij}$ can be build by inner products of an underlying set of Gram vectors $\phi_i$, where $i \in [1...N]$, i.e. $K_{ij} = \langle \phi_i, \phi_j \rangle$. We partition the matrix using (a) $l^2$ distances or (b) angles in Gram vector space. We use a similar notion to find row neighbors (for sampling and Near-far criterion).

We applied the approach altogether to 22 matrices from various physical domains – kernel-based matrices for simulations or data analysis, graph laplacians, and for wave and fluid transport problems. For almost all of them we observed good approximation behavior. We compared ourselves to the existing software packages STRUMPACK and HODLR that use the classical hierarchical semi-separability (HSS) approach. For some of the mentioned cases these codes cannot achieve required error tolerances due to missing direct evaluations (which we cover with our FMM approach). We show that for most cases we can achieve a significantly better performance than the two mentioned codes.

However, we currently only support shared memory parallelism (with optional GPU acceleration). Furthermore, we only optimized the evaluation phase because in applications this may be executed several times. STRUMPACK also offers distributed memory parallelism – we currently cannot compete with it regarding this feature.

Our software code GOFMM is developed along this thesis and we are planning to publish it soon. Along this thesis we attached an internal version.
5. Conclusion and future work

5.2. Future work

Gram vector space and low-rank approximability

In the course of this study we analyzed some synthetic cases where we specified the Gram vectors and looked at approximability. However, we questioned the actual Gram vector distribution for cases that allow far-field approximations. One could investigate further into that direction.

In [42] (Proposition 2.16, page 63), they use an essentially equivalent notion (the Cholesky columns) as the feature maps.

Target sampling

We have seen cases where we misestimate the approximation rank – and consequentially, we encounter very bad approximation accuracies. The problem is sparked by the nested interpolative decomposition. If we miss information (approximation behavior towards certain target points), we may select too few, or insufficient skeletons.

Hence, it is indispensable not to miss any “direction”. We experimented with other sampling techniques, such as a double pass or adding additional rows at uniform. Target sampling will be one of the predominant research topics for this study.

Matrix interface

Currently, for simplicity we achieve optimal performance if we allocate the matrix in memory. For dense cases we experimented with an out-of-core implementation, but this resulted in a major slow down. Currently, in order to achieve good performance for large matrices (that cannot be created on the fly) one needs big memory nodes. We do not touch all matrix entries, hence we want to optimize this approach by only allocating rows/columns that we actually need.

Alternatively, an interface to a matrix, that only requires to return the matrix-vector product (for any given vector) is a future goal.

Distributed memory parallelism

Additionally, a desirable goal is distributed memory parallelism. However, since we reorder the matrix it may be difficult regarding memory allocation. We can think of a single node out-of-core static tree computation, and a distributed compression and evaluation.

Matrix factorization

Hierarchical matrix codes often focus on acting as a solver – a feature which we do not support. Using the Sherman-Morrison-Woodbury formula and Schur complements one can derive a solver that allows fork blocks of direct evaluations (FMM).
Appendix
A. Detailed Descriptions

This section comprises a description of the source code – Python and C++ – datasets, and installation instruction. The source code is developed on private git repositories – we attached a CD along a hard copy of this thesis. If you happen to have an electronic version only – you are welcome to send me a short e-mail and I am happy to provide the source code for you. In the future we may publish it for all; the C++ repository has some other parts that are not open source.

A.1. Python code

A.1.1. Installation and Usage

As software dependencies we use numpy and scipy. If you do not have these packages installed already, e.g.:

```bash
>> pip install numpy
>> pip install scipy
```

Set testing conditions in `main.py`. Then execute:

```bash
>> python main.py
```

You may use the software in the full pipeline (see `main.py`) or for selected functions scripting style only. `main.py` should help in getting started.

A.1.2. File structure

`main.py` – sets the test case and runs the algorithms
`rkdt.py` – Nearest neighbor algorithms using rkdt; array as tree data structure
`tree.py` – Implements tree class and its algorithms
`test/` – folder with off_diag.py to try partitioning criteria
`test/off_diag.py` – script for partitioning only
`test/plot.py` – plot svd of level 1 off-diag matrix – `tryout.py` – script to create and store test matrices K `generate_matrices.py` – Kernel matrix creation

A.1.3. Test Matrices

In Folder “Matrices” test matrices are created with random Gaussian points in 3 dimensions, and a gaussian kernel function, $h = 0.1$.

In order to reproduce results you may also use the MATLAB script `spdmatrices.m` which stores matrix files binary (currently in single precision floating point format).

You can apply the software to any binary matrix files.
A. Detailed Descriptions

A.2. C++ Framework

GOFMM is developed in C++ (with c++11 features) and CUDA, employing OpenMP for shared memory parallelism using a self-contained runtime system.

A.2.1. Compilation

All CPU and KNL BLAS/LAPACK routines use MKL. GPU BLAS routines use CUBLAS; on ARM we use QSML (Qualcomm Snapdragon Math Library). KNL experiments use Cache-Quadrant configuration.

Given the repository url, you should be able to clone the master branch of the repository. The first step is to edit set.env.sh to select the proper compiler and architecture.

```bash
>> export XXXX_USE_INTEL = true
>> export XXXX_USE_INTEL = false
>> export XXXX_USE_CUDA = true
>> export XXXX_MIC_AVX512 = true
```

If user want to compile the CUDA code for the hybrid CPU-GPU implementation then the following variables have to be exported.

```bash
>> export XXXX_GPU_ARCH_MAJOR=gpu
>> export XXXX_GPU_ARCH_MINOR=pascal
```

There are three options for the host (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.

```bash
>> export XXXX_ARCH_MAJOR=ARM
>> export XXXX_ARCH_MINOR=ARMv8a
>> export XXXX_ARCH_MAJOR=x86_64
>> export XXXX_ARCH_MINOR=haswell
>> export XXXX_ARCH_MAJOR=mic
>> export XXXX_ARCH_MINOR=knl
```

Although we use cmake to identify BLAS/LAPACK libraries, but we suggest that user manually setup the path using
Finally, users must setup the OpenMP option to enable parallel implementation. Here for example, we use 68 threads for KNL and spread OpenMP thread binding.

```
>> export OMP_PROC_BIND=spread
>> export OMP_NUM_THREADS=68
```

With all these options setup, now we use cmake for compilation. Users can use the following commands.

```
>> source set_env.sh
>> mkdir build
>> cd build
>> cmake ..
>> make
>> make install
>> cd bin
>> ./run_gofmm_x86
>> ./run_gofmm_gpu
>> ./run_gofmm_knl
```

### A.2.2. Usage

Notice that the user interface of GOFMM is not yet finalized. Here we briefly introduce the interface of our test suit, and provide the prototype of two main routines (compress and evaluate). Here we demonstrate the example code that compresses an $5000 \times 5000$ random SPD matrix that is generated by our routines.

```c
/** type */
using T = float;
Data<T> *X = NULL;
SPDMatrix<T> K;
/** parameters */
//const SplitScheme SPLIT = SPLIT_RANDOM;
//const SplitScheme SPLIT = SPLIT_LEXIGRAPHICAL;
```
A. Detailed Descriptions

//const SplitScheme SPLIT = SPLIT_POINT_DISTANCE;
//const SplitScheme SPLIT = SPLIT_KERNEL_DISTANCE;
const SplitScheme SPLIT = SPLIT_ANGLE;
const bool ADAPTIVE = true;
const bool LEVELRESTRICTION = false;
std::string filename = std::string( "K02N65536.bin" );
std::size_t n = 65536;
std::size_t m = 512;
std::size_t k = 32;
std::size_t s = 512;
T tau = 1E-5;
std::size_t r = 512;
/** if coordinates are provided */
std::size_t d = 2;
std::string pointfilename = std::string( "X2DN65536.points.bin" );
X = new Data<T>( d, n, pointfilename );
/** read the matrix */
K.resize( n, n );
K.read( n, n, filename );
/** provide neighbors if any, otherwise using ANN */
Data<std::pair<T, std::size_t>> NN;
gofmm_setup<ADAPTIVE, LEVELRESTRICTION, SPLIT, T>(
    X, K, NN, n, m, k, s, tau, r );

A.2.3. Evaluation and expected result

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 runtime and total FLOPS of the compression and evaluation phase. Finally, the accuracy $\epsilon_2$ 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 success, the device name and the available global memory size should be displayed. The rest of the execution is the same as our architectures.
Bibliography


