A Kernel-Independent FMM in General Dimensions

William B. March
Institute for Computational Engineering and Sciences
The University of Texas
Austin, TX, USA
march@ices.utexas.edu

Bo Xiao
Institute for Computational Engineering and Sciences
The University of Texas
Austin, TX, USA
bo@ices.utexas.edu

Sameer Tharakan
Institute for Computational Engineering and Sciences
The University of Texas
Austin, TX, USA
sameer@ices.utexas.edu

Chenhan D. Yu
Department of Computer Science
The University of Texas
Austin, TX, USA
chenhan@cs.utexas.edu

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

We introduce a general-dimensional, kernel-independent, algebraic fast multipole method and apply it to kernel regression. The motivation for this work is the approximation of *kernel matrices*, which appear in mathematical physics, approximation theory, non-parametric statistics, and machine learning. Existing fast multipole methods are asymptotically optimal, but the underlying constants scale quite badly with the ambient space dimension. We introduce a method that mitigates this shortcoming; it only requires kernel evaluations and scales well with the problem size, the number of processors, and the ambient dimension—as long as the intrinsic dimension of the dataset is small. We test the performance of our method on several synthetic datasets. As a highlight, our largest run was on an image dataset with 10 million points in 246 dimensions.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Efficiency, Algorithm design and analysis; G.3 [**Probability and Statistics**]: Non-parametric statistics

Keywords

Machine learning, kernel methods, N-body problems, parallel algorithms, data mining, kernel ridge regression

1. INTRODUCTION

Problem definition. Given a set of N points $\{x_j\}_{j=1}^N \in \mathbb{R}^d$ and weights $w_j \in \mathbb{R}$, we wish to compute

$$u_i = u(x_i) = \sum_{j=1}^{N} \mathcal{K}(x_i, x_j) w_j, \quad \forall i = 1 \dots N. \tag{1}$$

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807647

Here $\mathcal{K}()$, a given function, is the *kernel*. Equation (1) is the kernel summation problem, also commonly referred to as an *N-body problem*. From a linear algebraic viewpoint, kernel summation is equivalent to approximating $u = Kw$ where u and w are N-dimensional vectors and K is the ***kernel matrix***, an $N \times N$ matrix consisting of the pairwise kernel evaluations. Direct evaluation of the sum requires $\mathcal{O}(N^2)$ work. Fast kernel summation can reduce this cost dramatically. For $d = 2$ and $d = 3$ and for a wide class of kernels, one can evaluate (1) in $\mathcal{O}(N \log N)$ work (treecodes) or $\mathcal{O}(N)$ work (fast multipole methods and fast Gauss transform methods) to arbitrary accuracy [12, 13].

The main idea in accelerating (1) is to exploit low-rank blocks of the matrix K. Hierarchical data structures reveal such low-rank blocks by rewriting (1) as

$$u_i = \sum_{j \in \text{Near}(i)} K_{ij} w_j + \sum_{j \in \text{Far}(i)} K_{ij} w_j, \tag{2}$$

where $\text{Near}(i)$ is the set of points x_j for which K is evaluated exactly and $\text{Far}(i)$ indicates the set of points x_j for which K is approximated. The first term is often referred to as the *near field* and the second as the *far field* for the point x_i. Throughout, we refer to a point x_i for which we compute u_i as a *target* and a point x_j as a *source* with *weight* (or *charge*) w_j. Treecodes approximately evaluate the "$j \in \text{Far}(i)$" term in $\mathcal{O}(\log N)$ time per target; fast multipole methods (FMM) evaluate it in $\mathcal{O}(1)$ time per target.

Significance. The FMM[1] and treecodes were originally developed to accelerate the solution of partial differential equations in dimension $d \leq 3$ [14]. Now their applicability spans several scientific domains: they are used in geostatistics [5], non-parametric statistics [29], machine learning [11, 17], approximation theory [30], Gaussian process modeling [25], and data assimilation [18]. These problems are often characterized by unconventional kernels and high ambient dimension d.

The algorithms and theory for the complexity and accuracy of fast multipole methods hold true for any arbitrary dimen-

[1]We will be using the term "fast multipole methods" loosely to refer to all methods for which that the complexity of the evaluation phase is $\mathcal{O}(N)$. The fast Gauss transform is another example.

sion. However the constants in the complexity estimates for both error and time do not scale well with d. For example, the constants in the original FMM [12], the kernel-independent FMM [34], and the generalized fast Gauss transform [27] scale exponentially with d. Modified algorithms like those discussed in [15] have constants that scale polynomially in d, but those methods are kernel-specific and either too expensive or too inaccurate for large ambient dimension d [21].

In [22], we introduced `ASKIT` (Approximate Skeletonization Kernel-Independent Treecode) and in [24], we introduced the parallelization of `ASKIT` (summarized in §2). In those works we made three notable contributions: a novel *algebraic* way to approximate the far field through the use of nearest-neighbor information, a new pruning method, and more efficient algorithms for the construction of the necessary data structures. `ASKIT`'s most important property is that accurate far-field approximations can be constructed with complexity that scales linearly with d for any accuracy level—depending only on the analytic properties of the kernel and the *intrinsic dimension* d_{intr} of the dataset.[2] Another property of `ASKIT` is its kernel independence. The kernel function has to admit low-rank factorizations of blocks in Far(i) in Eqn. 2 but does not need to be the solution of a PDE or be symmetric, *e.g.* it can be a variable-bandwidth Gaussian function.

Contributions. To our knowledge, there exists no scalable FMM that is kernel independent and whose complexity constant in the construction of the far field approximation is scalable with d. Here we propose and implement such a method.[3] In detail, the novelty of this paper can be summarized as follows:

- We introduce `ASKIT-FMM`, an algorithm that scales well with d, is kernel-independent, and whose approximation of the far field scales linearly with d (provided that $d_{\text{intr}} \ll d$) in §2.2. Depending on the problem, the original `ASKIT` algorithm may require more than 10× more evaluations compared to our new FMM code (§4, Table 2) for the same accuracy. We discuss the complexity and accuracy of the scheme in §3.
- We introduce several algorithmic innovations to both `ASKIT-FMM` and the original `ASKIT`, resulting in improved performance for both: improved tree construction, adaptive far-field approximation, blocking and FLOP-optimized kernel evaluations for x86 architectures, and level restriction (explained in §2.3). Taken together, these algorithms boost the overall accuracy and performance of the method and result in 20× speedups over our previous work (§4, Table 3). In addition, the new code supports different source and target sets.
- We present scalability results for four different kernels and for several real and synthetic datasets. Also, we present the application of `ASKIT-FMM` to a supervised learning task: kernel regression for binary classification of medical image data. This problem features many of the characteristics found in real applications: it requires solving linear systems with the kernel matrix, performing cross-validation and regularization, and amortization of setup costs across several evaluations. These results are discussed in §4.

[2]Many high-dimensional data sets can be accurately captured by a smaller set of (unknown) features. We refer generically to the size of this feature set as intrinsic dimension. For example, the intrinsic dimension of a set of points distributed on a curve in three dimensions is one.

[3]Our code is available at:
http://padas.ices.utexas.edu/software

Limitations. Although we have mitigated the dimensionality problem, we have not resolved it. The crossover N for which `ASKIT-FMM` is faster than the direct $\mathcal{O}(N^2)$ method rapidly increases with d_{intr} and accuracy requirements. Another problem is that the construction of the far-field approximation is quite expensive. Also, the method is not effective for oscillatory kernels (e.g., Helmholtz kernels). Although, we report results in two and three dimensions, we do not advocate the use of `ASKIT-FMM` for such problems. In low dimensions, much more efficient methods exist for all the kernels we consider here.

Related work. In [22, 24], we extensively review the literature regarding classical and general dimensional N-body methods. The most relevant publications to this work are [8] in low dimensions and [15] and [19] in high dimensions. Further discussion on the merits and shortcomings of these methods can be found in [24]. An entirely different line of work for high-dimensional kernel summations is the Nystrom method [20, 31]. This method constructs global low-rank approximations of the kernel matrix K and can be very efficient; however, not all kernel functions result in matrices that have a global low-rank structure [23]. Its comparison with `ASKIT-FMM` merits a thorough study, which is beyond the scope of this paper.

2. METHODS

Notation. We use sets to index vectors and matrices, such as $u(S)$ to refer to the entries of vector u corresponding to entries in some set S. We consider a binary, balanced, space partitioning tree with m points per leaf node. We use $\mathcal{X}$ for the data set, α to refer to a tree node, $\mathcal{X}_\alpha$ to refer to the points in α, and $\mathcal{S}_\alpha$ to refer to its skeleton points (§2.1). We use $\tilde{w}$ for the skeleton weights and $\bar{u}$ for the skeleton potentials. We use $\mathtt{l}(\alpha)$ and $\mathtt{r}(\alpha)$ to refer to the children of α. We use $\mathcal{A}(\alpha)$ for the ancestors of α, MORTONIDS(S) for the Morton IDs of points in a set S, and SIBLINGS(S) for the siblings of nodes in S. We use $\mathcal{N}_i$ for the list of the κ nearest neighbors of point i and $\mathcal{N}_\alpha$ for the union of the neighbor lists of all of the points in α. We will use Near(i) for the set of tree nodes which are evaluated directly with a target i and Far(i) for those approximated.

2.1 ASKIT Review

`ASKIT`, first presented in [22] and in parallel in [24], is a general dimension, kernel-independent treecode method. Here, we briefly review this method before focusing on the new features added in this paper.

Outgoing representation. A key feature of a treecode is the *outgoing representation*—the method used to approximate the contribution of the far field in (2). The classical FMM constructs the representation from series expansions of the kernel function. `ASKIT` uses an outgoing representation which requires no prior knowledge of the kernel function and scales with d.

Consider a leaf node α, and let $G = \mathcal{K}(\mathcal{X} \setminus \mathcal{X}_\alpha, \mathcal{X}_\alpha)$ – *i.e.* G is the $(N-m) \times m$ block of K with columns corresponding to source points in α and rows corresponding to all points not in α. Our task is to efficiently compute a low-rank approximation of G. We refer to the construction of this

representation as *skeletonization*.

The rank-s *Interpolative Decomposition* (ID) [6] of G is a factorization

$$G \approx G_{\text{col}} P, \tag{3}$$

where G_{col} consists of s columns of G. In other words, the ID approximates the matrix in a basis of its columns. We refer to the source points which correspond to the columns of G_{col} as the *skeleton points* of α, $\mathcal{S}_\alpha$. We also compute the *skeleton weights* $\tilde{w} = Pw$.

Then, we compute the contribution of node α in $\text{Far}(i)$ for some target i as

$$\tilde{u}(i) = \mathcal{K}(x_i, \mathcal{S}_\alpha)\tilde{w} \tag{4}$$

in $O(s)$ work by representing the source charges and points with effective charges at only the skeleton points.

Random sampling to compute the ID. We compute the ID from a pivoted, rank-revealing QR factorization of G. However, for a single leaf node, this requires $\mathcal{O}(Nm^2)$ work, which is greater than simply computing Gw directly. In order to overcome this, we employ a randomized approach, in which we sample ℓ rows of G and use them to approximately skeletonize G in ℓm^2 work. But how should we sample rows?

Sampling uniformly at random is easy but inaccurate; importance sampling is accurate but expensive [20, 21]. We use a hybrid approach that borrows from both these approaches. We sample using the κ nearest neighbors of each point. We then construct a matrix G whose rows correspond to the ℓ closest neighbors to the source points being skeletonized. If additional samples are required, we choose them uniformly at random. See [21] for a more thorough exploration of this sampling scheme.

Nearest neighbors. The nearest-neighbors of all points can be computed exactly or approximately (in high d) using randomized projection methods [1, 7, 11, 32]. For simplicity, we consider the nearest neighbors to be an input to `ASKIT`, since they can be pre-computed by any method.

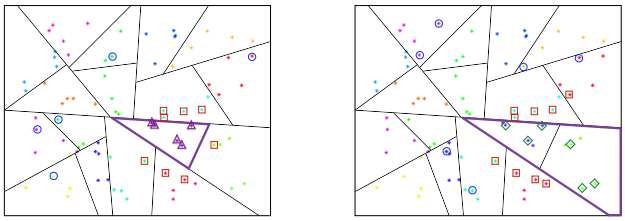
Outgoing-to-outgoing translation. Given the outgoing representations of two sibling nodes, we form the representation for their parent. We merge the two skeletons to obtain $2s$ points. We form another matrix G with $2s$ columns corresponding to these points, and we again collect rows from nearest neighbors and uniformly sampled points. We obtain s skeleton points (which are a subset of the child nodes' skeletons) and weights by computing an ID of this matrix. We illustrate this in Figure 1.

Pruning rule. `ASKIT` also uses a different method for the near-far field decomposition. Most treecodes use some explicit distance-based criterion for this split, typically based on the convergence of the series expansions used for the outgoing representation. While this is effective for low dimensional problems, it will not scale with increasing d.

Instead, `ASKIT` uses the nearest neighbors of each point to define the pruning rule. For each target, we assume we have its κ nearest neighbors. We prune a node if it does not own any of these neighbors.

Morton IDs. The Morton ID of a node is a bit string that encodes the path from the root to the node. The i^{th} bit of the Morton ID of a node is 1 if the node's ancestor at level i is the right child of its parent and 0 otherwise. We assign a point the same Morton ID as the leaf which owns it.

A node's Morton ID and level are sufficient to uniquely identify it in the tree. We collect the Morton IDs of the nearest neighbors of each target point. Then, we can construct



(a) **Leaf node.** (b) **Internal node.**

Figure 1: *We illustrate the skeletonization of nodes in ASKIT. In Figure 1(a), we show the skeletonization of a leaf node. The triangles highlight the points in the leaf, which correspond to the columns of G. We collect the nearest neighbors of these points, marked with squares. We also sample additional distant points uniformly at random (circles). These points correspond to the rows of G. We then compute an ID of this matrix to select skeleton points. For an internal node (Figure 1(b)), we merge the skeleton points and the neighbor lists of the children to form G. The resulting skeleton is a subset of the children's skeletons.*

the near and far field interaction lists:

$$\begin{aligned} \text{Near}(i) &= \text{MORTONIDS}(\mathcal{N}_i) \\ \text{Far}(i) &= \text{SIBLINGS}(\mathcal{A}(\mathcal{N}_i)) \setminus \mathcal{A}(\mathcal{N}_i) \end{aligned} \tag{5}$$

ASKIT. We see that the major components of `ASKIT` (Alg. 1) are: 1) *skeletonization* (line 3), in which we traverse the tree bottom-up to construct the skeleton of each node; 2) *list construction* (line 4) in which we form the interaction lists for each node; and 3) *evaluation* (line 5), in which we actually compute the kernel evaluations and the approximate potentials, using:

$$u(i) = \sum_{\alpha \in \text{Near}(i)} \mathcal{K}(x_i, \mathcal{X}_\alpha) w_\alpha + \sum_{\alpha \in \text{Far}(i)} \mathcal{K}(x_i, \mathcal{S}_\alpha)\tilde{w}_\alpha. \tag{6}$$

Algorithm 1 ASKIT(Dataset $\mathcal{X}$)

1: Read nearest neighbor info for all points in $\mathcal{X}$
2: Construct space partitioning tree α_{root}
3: **Skeletonize**(α_{root}) – Alg. 2
4: **ConstructInteractionLists** – Eqn. 5
5: **Evaluate** – Eqn. 6

Algorithm 2 Skeletonize(α)

1: `if` α is not a leaf
2: **Skeletonize**($\mathtt{l}(\alpha)$); **Skeletonize**($\mathtt{r}(\alpha)$)
3: $\mathcal{X}_\alpha = \mathcal{S}_{\mathtt{l}(\alpha)} \cup \mathcal{S}_{\mathtt{r}(\alpha)}$
4: $\mathcal{N}_\alpha = \left(\mathcal{N}_{\mathtt{l}(\alpha)} \cup \mathcal{N}_{\mathtt{r}(\alpha)}\right) / \mathcal{X}_\alpha$
5: Collect the closest ℓ points in $(\mathcal{N}_\alpha \cup \mathtt{uniform})$ into $\mathcal{T}_\alpha$
6: Form $G = \mathcal{K}(\mathcal{T}_\alpha, \mathcal{X}_\alpha)$ and compute ID $G_{\text{col}}P$
7: Store skeleton $\mathcal{S}_\alpha$ and skeleton weights $\tilde{w}_\alpha$

2.1.1 ASKIT in Parallel

Our parallel implementation of `ASKIT` uses a hybrid MPI / OpenMP scheme with p distributed processes, where p is a power of two.

For the construction of the space-partitioning tree, we refer the reader to [24, 33]. For the remainder of our discussion, we only point out that the parallel tree construction assigns N/p points to each process, which form a subtree. We refer to this subtree as the *local tree*. We refer to the portion of the tree above level $\log p$ as the *distributed tree*. After tree construction, each processor has the Morton IDs and coordinates of the nearest neighbors of all of its points.

Skeletonization. Within the local tree, we proceed in a level-by-level fashion from the leaves to the local root. The skeletonization of each node in a level is independent of all other nodes at that level, so these can be assigned to different threads. Skeletonization of the distributed tree is computed as a reduction from level $\log p$ up to the root.

Evaluation with a local essential tree. Each process computes the potential for all targets in its local tree, so it needs the contents of all nodes in $\text{Near}(i)$ and $\text{Far}(i)$ for every target i it owns.

We collect all of these nodes into a local essential tree [28]. Let $\mathcal{X}_p$ be the set of all points owned by process p and $\mathcal{N}_p$ be the set of all nearest neighbors of $\mathcal{X}_p$. Then, let $\mathcal{L}$ be the set of all leaves containing points in $\mathcal{N}_p \setminus \mathcal{X}_p$. The LET is the set of nodes

$$\text{LET} = \mathcal{L} \cup [\cup_{\alpha \in \mathcal{L}} [\text{SIBLINGS}(\mathcal{A}(\alpha)) \setminus \mathcal{A}(\alpha)]] \quad (7)$$

Since the Morton ID and level of a node uniquely identify it, and since each process has the Morton IDs of all of the nearest neighbors of its target points, it can identify locally the nodes it needs for its LET.

Each node can then obtain the skeletons and coordinates of these nodes in two communication phases. First, an all-to-all primitive allows each process to send its node requests to all other processes. Then, in a second all-to-all, each process answers these requests with either the point coordinates and charges (for a leaf node), or skeleton points and skeleton weights (for an approximated node). Once each process has the contents of its LET, it can evaluate its potentials in parallel over targets or blocked interaction lists (§2.3).

2.2 ASKIT-FMM

Here, we give our new extension of `ASKIT` to an FMM-like algorithm. In addition to the outgoing representation, which compactly represents a large group of source points, an FMM requires an incoming representation to summarize a group of target points and an outgoing-to-incoming translation between nodes.

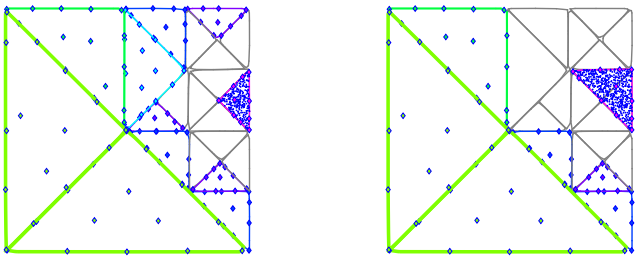
Incoming representation. As before, we require a low-rank approximation of a matrix $G = \mathcal{K}(\mathcal{X}_\alpha, \mathcal{X} \setminus \mathcal{X}_\alpha)$—the $m \times (N-m)$ matrix of interactions between all targets in a leaf node α and all distant sources. We use the ID to build an incoming representation as well, using rows of the matrix as the basis:

$$G \approx E G_{\text{row}}. \quad (8)$$

Note that when the kernel function is symmetric, this matrix is the transpose of the one approximated for the outgoing representation. In this case, we use the same points for both skeletons, and use $E = P^T$[4].

Then, for target points in α and some source points x_i, we compute *skeleton potentials* at the incoming skeleton points

[4] For non-symmetric kernels, we need to perform two interpolative decompositions for each node, one for each of the incoming and outgoing representations.



(a) **Leaf.** (b) **Internal node.**

Figure 2: *We illustrate the FMM interaction lists. In Fig. 2(a), we highlight in green and blue colors the nodes in the list* FMMFar *for the leaf node in magenta with blue target points. The interactions between it and all of the highlighted nodes can be approximated by only computing kernel interactions between its skeleton points and the skeleton points of the green and blue nodes. In Fig. 2(b), we show the list* FMMFar *for the parent of the target leaf in 2(a). Since all the points in both children can prune the nodes highlighted in green and blue, they can interact approximately with the skeleton of the magenta parent. The resulting skeleton potentials will be passed down with Alg. 4.*

due to some set of source points as

$$\bar{u}(\mathcal{S}_\alpha) = \sum_j \mathcal{K}(\mathcal{S}_\alpha, x_j) w_j. \quad (9)$$

We can then obtain the approximate potentials at the remaining target points by computing $u = P^T \bar{u}$.

FMM interaction lists. In `ASKIT-FMM`, the skeletonization phase proceeds exactly as in the treecode version (Alg 2). The differences begin with the formation of the interaction lists. In a classical FMM, target nodes only interact with source nodes at the same level. Since `ASKIT` uses an irregular, combinatorial pruning criterion, we use a different approach.

We begin by forming the interaction lists $\text{Far}(i)$ for each target i in (5), which can be done in parallel. We then merge the lists from leaves up the tree, once again in parallel within each level (as in skeletonization). Concretely, for a leaf node α, we compute a list[5]

$$\text{FMMFar}(\alpha) = \cap_{i \in \alpha} \text{Far}(i). \quad (10)$$

We then remove the merged nodes from each target list:

$$\text{Far}(i) = \text{Far}(i) \setminus \text{FMMFar}(\alpha). \quad (11)$$

We show this method for both leaves and internal nodes in Alg. 3 and illustrate it in Figure 2.

Outgoing-to-incoming translation. We now modify the evaluation phase described in (14). The evaluation for $\text{Near}(i)$ and $\text{Far}(i)$ proceeds exactly as before. For the new list FMMFar, we compute an incoming representation (skeleton potentials for the skeleton of node α) from the outgoing representation of node β in $\text{FMMFar}(\alpha)$ as

$$\bar{u}(\mathcal{S}_\alpha) \mathrel{+}= \mathcal{K}(\mathcal{S}_\alpha, \mathcal{S}_\beta) \tilde{w}_\beta. \quad (12)$$

Incoming-to-incoming translation. We obtain the final potentials through an additional top-down pass through

[5] This list is similar to the *V-list* in the classical FMM, but it can contain nodes at many different levels of the tree.

the tree (Alg. 4). We apply P^T to the skeleton potentials for each internal node to obtain the skeleton potentials for the child nodes. At the leaf level, an additional application of P^T gives the potential. Once again, this step can be performed in parallel across different nodes in the same level of the tree.

ASKIT-FMM in parallel. `ASKIT-FMM` requires no additional communication for a parallel implementation. The merging of interaction lists is done locally on each process across each level and in parallel within a level. The evaluation phase only requires skeletons and points that are in the treeecode version LET, so we can re-use the same approach for the FMM—*i.e.* the LET's for `ASKIT` and `ASKIT-FMM` are the same requires no change. The skeleton potentials are only passed down locally, so this requires no extra communication as well and can again be done in parallel within each level. Our current implementation does not merge interaction lists for nodes in the distributed tree.

Algorithm 3 FMMInteractionLists(Dataset $\mathcal{X}, \alpha_{\text{root}}$)

```
parfor all target points i
    Near(i) = MORTONIDS(N_i)
    Far(i) = SIBLINGS(A(N_i)) \ A(N_i)
parfor all leaves α
    FMMFar(α) = ∩_{i∈α} Far(i)
    Far(i) = Far(i) \ FMMFar(α)
for all non-leaves α // in parallel on each level
    FMMFar(α) = FMMFar(l(α)) ∩ FMMFar(r(α))
    FMMFar(l(α)) = FMMFar(l(α)) \ FMMFar(α)
    FMMFar(r(α)) = FMMFar(r(α)) \ FMMFar(α)
```

Algorithm 4 IncomingToIncoming(tree α_{root})

```
for level ℓ from 0 to log(N/m) of α_root
    parfor all nodes α at level ℓ
        if α is not a leaf
            [ū(l(α)), ū(r(α))] += P_α^T ū(α)
        else  u(α) += P^T(α) ū(α)
```

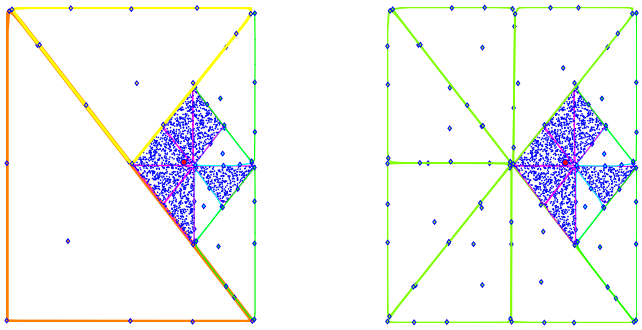
2.3 ASKIT Improvements

We now turn to our other improvements to `ASKIT` and `ASKIT-FMM` introduced in this paper.

Adaptive rank skeletonization. Previously, we chose the skeleton size s as an input parameter. The new algorithm estimates the singular values of the off-diagonal block G and uses them to choose an approximation rank.

We specify a rank tolerance parameter τ. When skeletonizing, we construct the subsampled off-diagonal block G as above. We compute a rank-revealing QR factorization $G\Pi = QR$. We then employ the diagonal entries R_j of R as an estimate for the spectrum of G. We set $s = \arg\min_j |R_{j+1}/R_1| < \tau$. We also specify a maximum rank $s_{\max}$. If the adaptively selected rank is greater than $s_{\max}$, we use a rank $s_{\max}$ decomposition instead.

Level restriction. In the level restricted version of `ASKIT`, we do not construct skeletons or prune any nodes above tree level L (with the root being level 0). This increases the accuracy of `ASKIT`, since it uses more skeleton points per target point but at a higher cost. In effect, level restriction allows us to trade additional work for increased accuracy. `ASKIT-FMM` can reduce this extra cost as we will see in the results section. We illustrate this restriction in Figure 3.



(a) $L = 0$. (b) $L = 3$.

Figure 3: *We illustrate level restriction in* `ASKIT`. *We highlight a single target point in red, then show the evaluations required for two different values of the level restriction parameter L. We show skeleton points which interact with the target with diamonds and use different colors to indicate different levels. We show the direct evaluations with blue dots. Note that the level restricted version in Figure 3(b) interacts approximately with more nodes than in Figure 3(a), but each skeleton approximates fewer points. The direct interactions are the same in both cases.*

In terms of (5), we split any nodes in $\text{Far}(i)$ that are above level L and add all of their descendants at level L to $\text{Far}(i)$.

Reduced neighbor and skeleton communication. In the previous version of `ASKIT`, we exchanged the coordinates for the skeleton points for each node in the LET. However, each skeleton point at some level in the tree also appears as a skeleton point at each level below. In practice, the LET will often contain many of these nodes. Our updated implementation only exchanges and stores one copy of the coordinates of any skeleton point, thus reducing the storage and communication costs, particularly when d is large. Additionally, our previous implementation maintained a separate copy of the coordinates of nearest neighbors and points communicated in the LET. We have eliminated this extra storage, allowing our LET code to scale much better with larger values of κ.

Blocking kernel evaluations. In [24], we introduced an efficient method for computing kernel interactions based on blocking the interaction lists. Concretely, for each source node α, we compute lists

$$\begin{aligned} \text{Near}(\alpha) &= \{i : \alpha \in \text{Near}(i)\} \\ \text{Far}(\alpha) &= \{i : \alpha \in \text{Far}(i)\} \end{aligned} \tag{13}$$

The evaluation phase then consists of iterating over all nodes α and computing:

$$\begin{aligned} u(\text{Near}(\alpha)) &\mathrel{+}= \mathcal{K}(\text{Near}(\alpha), \mathcal{X}_\alpha) w_\alpha \\ u(\text{Far}(\alpha)) &\mathrel{+}= \mathcal{K}(\text{Far}(\alpha), \mathcal{S}_\alpha) \tilde{w}_\alpha \end{aligned} \tag{14}$$

In our previous work, we explored the efficient evaluation of the inverted near-field list on a GPU, while we computed the far-field list on the CPU. In this paper, we take advantage of the fact that interactions in all three lists are simply kernel matrix-vector products. We merge these lists into a single interaction list and employ a new, efficient kernel summation library to carry out these products [35]. While the details will be presented elsewhere, we mention that the kernel evaluations are heavily optimized for x86 architectures.

The blocked calculation over $\text{Far}(\alpha)$ and $\text{FMMFar}(\alpha)$ re-

quires collecting the skeleton weights $\tilde{w}$ into contiguous memory. We store the tree nodes in a pre-order traversal. After the skeletonization phase, we use a parallel scan to allocate space for these weights, then another parallel loop to store them in the allocated memory. We also need to allocate additional contiguous storage for the skeleton potentials in `ASKIT-FMM`.

Repeated applications of K. In practice, kernel methods often require the solution to a linear system using the kernel matrix. In the absence of a direct solution method, iterative methods are needed. `ASKIT-FMM` is well-suited to the repeated application of K to new right hand sides. Initially, we compute the skeleton of each node, construct the interaction lists, and exchange the LET.

We store the matrix P used in skeletonization. Given a new charge vector, we update the skeleton weights in a bottom-up tree traversal (Alg. 5). (In the distributed tree, we can collect potentials from children in a distributed reduction.) At each node, we compute a matrix-vector product Pw_α. Note that we do not have to compute any additional QR factorizations, which are the dominant cost for the skeletonization phase.

Algorithm 5 UpdateCharges(tree α_{root})

1: `for` level ℓ from $\log(N/m)$ to 0 of α_{root}
2: `parfor` all nodes α at level ℓ
3: `if` α is not a leaf
4: $w_\alpha = [\tilde{w}(\mathtt{l}(\alpha)), \tilde{w}(\mathtt{r}(\alpha))]$
5: $\tilde{w}(\alpha) = P_\alpha w_\alpha$

We then need to communicate the new source and skeleton charges to each node which has them in its LET. This communication complexity is similar to that for the original LET, but lacks the factor of d since no coordinates need to be communicated. Finally, we simply compute the evaluation phase in (14). We can store the blocked interaction lists between interactions, since they require no updates.

Distinct target and source sets. So far, we have assumed that the source and target sets are the same. Since we apply `ASKIT-FMM` to supervised learning tasks, we consider the case where we first use the same target and source sets (the *training phase*), then add a new target set and compute potentials for points in it (the *testing phase*).

The training phase proceeds exactly as in Alg. 1. We then load the new test set and the κ nearest neighbors of each test point in the training set. We assign each test point the Morton ID of the leaf node that owns its nearest neighbor. For each new test point, we form interaction lists using the nearest neighbor list, exactly as in the training phase (5) and invert them as in (13).

We then update the LET with any nodes that were not previously part of any interaction list. Note that this step is not required in classical FMM methods. In general, however, most of the nodes needed in the test LET will already have been obtained for the training LET. We only communicate any new nodes required. Adding new test points does not require any additional skeletonization.

`ASKIT-FMM` for testing points requires the ability to translate skeleton potentials on the skeleton points of a leaf node to the testing points in that node. While this mapping can be constructed from an additional ID, we leave the discussion of this method to future work.

3. THEORY

We summarize the existing complexity and error results for `ASKIT` and present new results for `ASKIT-FMM`.

`ASKIT-FMM` error. Here we use some additional notation: $\mathcal{D} = \log N/m$ for the tree depth and $n = N/p$ for the number of points per MPI process. We derived error bounds for `ASKIT` in [22], and now extend them to the `ASKIT-FMM`. Due to space limitations we cannot present the details of the derivation here. The results below should be interpreted as worst-case estimates.

`ASKIT` constructs low-rank approximations of off-diagonal sub-blocks of K (§2.1), denoted by G. The accuracy of the approximation is mainly controlled by the rank used in the ID and the sampling error. The former depends on the decay of the singular values of G [6, 16] and is controlled by s and τ in our algorithm. The sampling error is controlled by the sampling size and the quality of the samples. Increasing the number of nearest neighbors improves the quality of the samples chosen to form the ID and increases the number of direct evaluations performed—both of which result in higher accuracy and higher costs. `ASKIT-FMM` has one additional possible source of error—the approximation of the incoming representation. We can show that the overall error bound does not change, only the constant prefactors do. Essentially, if σ_{s+1} is the largest error incurred in the low-rank approximation of any G, then the error behaves as

$$\|K - K_{\mathtt{askit-fmm}}\|_2 \leq \ \sigma_{s+1}\mathcal{D}. \tag{15}$$

`ASKIT` complexity. Here we summarize the results found in [22, 24]; t_s will be the latency, t_w the reciprocal of the bandwidth; and we will assume a hypercube topology and that a single kernel evaluation requires $\mathcal{O}(d)$ work.

We first construct the tree in parallel, then we create the skeletonization (outgoing and incoming representations), then construct the local essential tree, and finally we evaluate in an embarrassingly parallel manner.

- *Parallel tree construction* (without the LET) was discussed in detail in [32, 33] and requires

$$(t_s + t_w)\log^2 p \log N + (t_w \log p)(d+\kappa)\, n$$

time. The storage is $\mathcal{O}(d\kappa n)$.

- *Skeletonization* requires $\mathcal{O}(ds^2 + s^3)$ work for an internal node and $\mathcal{O}(dsm + sm^2)$ work for a leaf (with $\ell = \mathcal{O}(s)$). Nodes deeper than $\log p$ level require no communication; closer to the root we use a reduction in which we combine the skeletons of two sibling nodes by exchanging a $\mathcal{O}(ds)$-length message. Thus, the communication is bounded by $\mathcal{O}((t_s + t_w)sd \log p)$. Since we have a total of $2(n/m + \log p)$ nodes per MPI process, and assuming that $m \leq s$ (to simplify the expressions), the total skeletonization time is

$$\left(\frac{n}{m} + \log p\right)(ds^2 + s^3) \tag{16}$$

and the storage cost is $s^2 n + (ds + s^2)\log p$.

- *LET construction* complexity estimates are based on the fact that for any target i, the number of nodes in the near and far interaction lists is given by

$$\begin{array}{rcl} |\text{Near}(i)| & = & \mathcal{O}(\kappa) \\ |\text{Far}(i)| & = & \mathcal{O}(\kappa\mathcal{D}). \end{array} \tag{17}$$

In the worst case, every node in Near(i) and Far(i) for every target point i must be received from another MPI process.

Thus, the worst case of the communication during the LET exchange is

$$t_s p + t_w d\kappa(m + s\mathcal{D})n. \tag{18}$$

The LET requires additional storage of size $d\kappa\,(m + s\mathcal{D})\,n$. Note that these bounds are pessimistic if $d_{\text{intr}} \ll d$ since they ignore overlaps between the lists.

• *Evaluation* is embarrassingly parallel once we have the LET. Its time complexity is given by

$$dn\kappa(m + s\mathcal{D}). \tag{19}$$

Complexity of `ASKIT-FMM`. The construction and skeletonization phases are identical to `ASKIT`'s. What differs is the construction of the FMMFar for each node and the evaluation. To construct the FMMFar lists for every node we use Alg. 3, which requires merging m lists of size $\mathcal{O}(\kappa\mathcal{D})$ at the leaves and then a bottom-up recursive merging of the lists of siblings. The work is $\mathcal{O}(\kappa\mathcal{D}N)$ and is part of the setup phase.

During the evaluation phase, we first convert the outgoing representations to incoming representations incurring a ds^2 cost per entry of FMMFar. Passing the skeleton potentials down the tree (Alg. 4) is an s^2 calculation per node. Finally, for every target point i, we perform s evaluations for every node in Far(i) that could not be merged into the FMMFar of the leaf that contains i. Thus, the total evaluation cost comprises the cost of the direct interactions plus the outgoing-to-incoming interactions plus the outgoing-to-target interactions. The last two terms are hard to estimate. To simplify the discussion, we introduce ζ as percentage of evaluations done using the outgoing-to-incoming interactions and obtain

$$dn(\kappa m + \zeta s^2/m + (1-\zeta)s\kappa\mathcal{D}) \tag{20}$$

for the time complexity of the evaluation, where the first term is from nodes in Near, the second is from nodes in FMMFar, and the third is from nodes in Far that could not be merged.

On one hand, if $\kappa = 1$, pruning is perfect. Every target has its own leaf node in Near and all the siblings of its ancestors in Far. In this case, the FMM interaction list FMMFar(α) always consists of the sibling of α. Thus, $\zeta = 1$ and the overall evaluation cost is $\mathcal{O}(nd(\kappa m + s^2/m))$.

On the other hand, there may be no overlap between the interaction lists for the targets in a node. This can happen when both d_{intr} and κ are very large, for instance. In this case, we may not be able to remove any entries from Far, leaving FMMFar empty and resulting in $\zeta = 0$. However, the evaluation phase in this case is no more expensive than in the treecode version of `ASKIT` in (19).

As a third case, consider a classical convergence analysis of fast multipole methods, but with a twist: assume $d_{\text{intr}} \ll d$. In d_{intr} dimensions, every node has $\approx 3^{d_{\text{intr}}}$ nodes nearby. Thus, we expect $|\text{FMMFar}| = \max(0, \kappa m(\mathcal{D} - 3^{d_{\text{intr}}}))$ and one can generalize this expression to interior nodes. In this case, for any d_{intr} as N increases, the size of FMMFar will grow, ζ will tend toward one, and we obtain an $\mathcal{O}(N)$ method. This result is not surprising as it echoes known FMM results in d_{intr} dimensions. The main point is that unlike the classical FMM, `ASKIT-FMM` "sees" only d_{intr}, not d. `ASKIT-FMM` does not require that d_{intr} is flat or on a manifold, although the presence of a high curvature complicates the analysis since it introduces scale effects.

• *Level restriction:* When we restrict the level to L, all the formulas remain the same, we only have to replace $\mathcal{D}$ with $\mathcal{D} - L + 2^L$. Equation (15) for the error becomes $\sigma_{s+1}(\mathcal{D} - L)$, with a smaller σ_{s+1} since the maximum errors occur near the root where skeletons approximate more points. The error improves but the time complexity increases and the LET requires more storage.

Next, we report experimental results that give a sense for the behavior of our method in high dimensions and in real applications.

4. EXPERIMENTS

We study the performance of the new `ASKIT` and `ASKIT-FMM` for different kernels, parameters, and datasets. To demonstrate the workflow and overall costs of applying `ASKIT-FMM` to an application, we report results for kernel regression.

Dataset			kNN			
name	**N**	**d**	κ	**iter**	**hit rate**	**Error**
SUSY	4,500,000	18	2,048	100	98%	8E-4
HIGGS	10,500,000	28	2,048	100	91%	3E-2
BRAIN	10,584,046	246	2,048	31	>99%	4E-4

Table 1: *The data sets used in our experiments. **N** is the training set size and **d** is the dimension. We give the number of iterations, hit rate, and error for our approximate nearest neighbor search. See [33] for details.*

Datasets and machines. (Table 1) We use points sampled from a uniform distribution in the unit hypercube in 2D and 3D ("Uniform"). We also uniformly generate points in 6D and then embed them in 64D ("Uniform 64D"). From high-energy physics, we use the "HIGGS" and "SUSY" sets from the UCI Machine Learning repository [2]. These sets correspond to a learning task to classify high-energy particles [4]. We also use a set of features obtained from brain medical images. Since, `ASKIT` needs nearest neighbors, for each dataset we report the number of iterations and neighbor distance accuracy. The nearest neighbors were computed with the randomized KD-tree search scheme described in [33].

Setup. All tests took place on the Maverick and Stampede clusters at the Texas Advanced Computing Center. Maverick nodes have two Intel Xeon E5-2680 v2 (2.8GHz) CPUs and 256GB RAM. Stampede nodes have two Intel Xeon E5-2680 (2.7GHz) CPUs and 32GB RAM. All tests were done in double-precision arithmetic. Our `C++` implementation uses OpenMP, MPI, MKL, and vectorization using x86 intrinsics.

Kernels. We show results for the following kernel functions $\mathcal{K}(x, y)$ (omitting normalization constants):

Gaussian	$\exp\left(-\|x-y\|^2/(2h^2)\right)$
Laplace	$\|x-y\|^{2-d}$
Matern	$\left(\sqrt{2\nu}\|x-y\|\right)^{\nu} K_\nu\left(\sqrt{2\nu}\|x-y\|\right)$
Polynomial	$\left(x^T y/h + c\right)^p$

The Gaussian is parameterized by the bandwidth h. The Laplace is valid for $d > 2$. The Matern kernel [30] is parameterized by ν; K_ν is the modified Bessel function of the second kind. The polynomial kernel is parameterized by degree p, bandwidth h, and constant c. We use an optimized implementation [35] for the Gaussian and Laplace kernel functions. Our Matern kernel implementation uses the GSL library [10]

		Uniform 2D			Uniform 64D (6ID)			SUSY	HIGGS
#	κ	1M	4M	16M	1M	4M	16M	4.5M	10.5M
					L = 4				
1	1	14%	13%	11%	14%	13%	11%	12%	12%
2	32	38%	35%	32%	89%	85%	81%	91%	>99%
3	128	42%	38%	36%	94%	91%	88%	95%	>99%
					L = 9				
4	1	53%	13%	4%	53%	13%	4%	12%	5%
5	32	53%	15%	5%	62%	15%	16%	32%	76%
6	128	54%	15%	5%	66%	15%	21%	39%	86%

Table 2: ***Kernel independent experiments.*** *We show the fraction of kernel evaluations done in ASKIT-FMM compared to the treecode for several data sets and values of κ. The column "#" labels rows for reference in the text. Here, we fix the skeleton rank $\mathbf{s = 1024}$ and leaf size $\mathbf{m = 128}$. The first block of results correspond to $\mathbf{L = 4}$ and the second to $\mathbf{L = 9}$. In these experiments, the interaction lists are completely kernel independent.*

#	Alg.	ϵ_r	T_{skel}	T_{list}	T_{LET}	T_E
7	IPDPS	1E-01	18	1	5	20
8	FMM	6E-02	11	6	8	6
9	FMM, Adaptive	1E-01	1	6	4	1

Table 3: *We compare results on the **SUSY** data between our work and [24], Table IV, last row. The column "#" labels rows for reference in the text, "**Alg**" identifies the treecode or FMM variant. The remainder of the columns are timings in seconds: T_{skel} is the skeletonization time, T_{list} is the time to form, invert, and merge the interaction lists, T_{LET} is the time to construct and exchange the LET, and T_E is the time for the evaluation phase. All experiments use a Gaussian kernel with $\mathbf{h = 0.15}$ and $\mathbf{m = 512}$, $\mathbf{s = 512}$, $\kappa = \mathbf{64}$, and $\ell = \mathbf{s}$. The charges are all $1/\sqrt{N}$ and the reported error is the relative error $|u - \tilde{u}|/|u|$ averaged over 1,000 evaluation points. The adaptive rank experiment uses $\tau = \mathbf{0.1}$. All experiments use 32 nodes of Maverick with 2 MPI processes per node and 10 OpenMP threads per process.*

#	Alg	L	ϵ_2	T_E	#K
10	Tree	2	1E-05	33	68,382
11	FMM	2	8E-06	34	65,119
12	Tree	4	1E-06	40	87,159
13	FMM	4	1E-06	37	71,084
14	Tree	6	3E-07	80	178,131
15	FMM	6	3E-07	46	88,311
16	Tree	8	4E-08	257	561,957
17	FMM	8	5E-08	73	143,931

Table 4: *Experiments for **Uniform 64D (6 intrinsic), $\mathbf{N = 16E6}$ data** and Gaussian kernels with varying level restrictions. The column "#" labels rows for reference in the text, "**Alg**" identifies the treecode or FMM variant, "**L**" is the level restriction parameter, T_E is the evaluation time in seconds, and **#K** is the number of kernel evaluations per target. We use a Gaussian kernel with $\mathbf{h = 0.7}$ and set $\kappa = \mathbf{32}$, $\mathbf{m = 512}$, and $\mathbf{s = 2048}$ in all experiments. For the experiment in row 11, the skeletonization time is 3500s, list construction takes 47s, and the LET exchange takes 29s. All experiments are on 16 nodes of Maverick with 20 OpenMP threads per node.*

to evaluate the Bessel functions and BLAS to compute kernel matvecs. Our polynomial kernel implementation uses BLAS for both steps.

Kernel matvec approximation experiments. We measure the accuracy and time of `ASKIT` and `ASKIT-FMM`. We are interested in the accuracy of our approximated potential $\tilde{u} = \tilde{K}w$ for a given charge vector w. We report the relative error $\epsilon_2 = \|Kw - \tilde{K}w\|_2/\|Kw\|_2$. Since the exact Kw is prohibitively expensive, we sample $n_s = 1,000$ entries of u and report ϵ_2 for this subset. Unless otherwise noted, we use charges w drawn from a standard normal distribution and average our results over 10 independent charge vectors. To compare with [24], in Table 3 we report $\epsilon_r = n_s^{-1} \sum_i |u_i - \tilde{u}_i|/|u_i|$. Next we discuss our results. We use the run ID ("#" in the tables) to identify a run. In all experiments except Table 3, we use the improvements described in §2.3 for both `ASKIT` and `ASKIT-FMM`.

Comparison to [24]. We present a brief comparison between the `ASKIT-FMM` and the implementation of `ASKIT` used in Table 3. Row 7 is taken directly from the last row of Table IV in [24] and does not employ any of the optimizations described in §2.3. We compare it to `ASKIT-FMM` with fixed rank (#8) and adaptive rank (#9). `ASKIT-FMM` delivers a drastically faster evaluation phase (20x faster for #9 with no increase in error). The list construction is somewhat unoptimized and it shows. The LET exchange is slightly slower in #8 than in #7, but #7 requires more storage. The adaptive skeletonization in #9 is much faster because the QR factorizations required are smaller.

Kernel evaluations in FMM. Another important issue the number of kernel evaluations when switching from the treecode to the FMM. In Table 2, we study this effect as a function of L, κ, and the dataset. We use fixed rank s. These results hold for any kernel function since they only depend in the s and κ parameters.

We see that increasing κ always requires more interactions for `ASKIT-FMM`. This is because the larger neighbor lists result in decreased opportunities for list merging in Alg. 3. Similarly, we see that increasing d also harms the performance of `ASKIT-FMM`. Both of these effects relate to our discussion in §3: as d increases, there is likely to be less overlap between the neighbor lists of nearby target points. Increasing the level restriction L always helps the FMM version, except for the smaller data sets. For these, `ASKIT-FMM` requires more kernel evaluations because up to level 9 of the tree, there is no compression. (These node-to-node interactions are equivalent to the point-to-node interactions in the treecode).

Level restriction. We explore the effect of level restriction in more detail in Table 4. In general, we see two clear

#	Alg	κ	L	m	τ	T_E	#K
Gaussian, $\mathbf{h = 0.1}, \epsilon_2 = \mathbf{1e\text{-}5}$							
18	Tree	32	4	512	1E-12	0.57	24,904
19	FMM	32	4	512	1E-12	0.27	4,869
Gaussian, $\mathbf{h = 0.05}, \epsilon_2 = \mathbf{1e\text{-}3}$							
20	Tree	32	4	512	1E-12	0.86	42,579
21	FMM	32	4	512	1E-12	0.53	11,177
Laplace, $\epsilon_2 = \mathbf{8E\text{-}3}$							
22	Tree	2	10	512	1E-15	142	878,290
23	FMM	2	10	512	1E-15	39	198,003
Matern, $\nu = \mathbf{1.2}, \epsilon_2 = \mathbf{8E\text{-}8}$							
24	Tree	32	4	512	1E-12	1.99	2,445
25	FMM	32	4	512	1E-12	2.02	1,540
Polynomial, $\mathbf{p = 2}, \epsilon_2 = \mathbf{1E\text{-}13}$							
26	Tree	1	4	128	1E-12	3.05	382
27	FMM	1	4	128	1E-12	0.56	124

Table 5: *Results for **Uniform 3D data,** $\mathbf{N = 4E6}$ and different kernel functions. The column "#" labels rows for reference in the text, "**Alg**" identifies the treecode or FMM variant, $\boldsymbol{T_E}$ is the evaluation time in seconds, and #$\boldsymbol{K}$ is the number of kernel evaluations per target. We use $\mathbf{s}_{max} = \mathbf{2048}$ for all the experiments. The Matern kernel experiments were run on Stampede; the others were on Maverick. The polynomial kernel experiment uses $\mathbf{h = 1}$ and $\mathbf{c = 1}$. For #23, skeletonization requires 29s, list construction requires 183s, and the LET exchange requires 4s. The Maverick experiments use 20 OpenMP threads per node and the Stampede experiments use 16.*

#	Alg	p	T_E	#K
SUSY, $\epsilon_2 =$ 1E-3				
28	Tree	8	9.3	65,533
29	FMM	8	8.6	56,753
30	Tree	16	4.7	65,531
31	FMM	16	4.4	56,750
32	Tree	32	3.6	65,530
33	FMM	32	3.9	56,750
HIGGS, $\epsilon_2 =$ 9E-2				
34	Tree	32	204	2,917,630
35	FMM	32	265	2,693,860
BRAIN, $\epsilon_2 =$ 5E-2				
36	Tree	32	61	124,856
37	FMM	32	85	100,233

Table 6: *Experiments on **real data sets** with Gaussian kernels. The column "#" labels rows for reference in the text, "**Alg**" identifies the treecode or FMM variant, "**p**" is the number of compute nodes, "**L**" is the level restriction parameter, $\boldsymbol{T_E}$ is the evaluation time in seconds, and #$\boldsymbol{K}$ is the number of kernel evaluations per target. The SUSY experiments use $\kappa = \mathbf{1024}$, $\mathbf{m = 1024}$, $\mathbf{h = 0.05}$, $\mathbf{s}_{max} = \mathbf{2048}$, $\tau = \mathbf{0.5}$, and $\mathbf{L = 5}$. The HIGGS experiments use $\kappa = \mathbf{1024}$, $\mathbf{m = 512}$, $\mathbf{h = 1}$, $\mathbf{s}_{max} = \mathbf{2048}$, $\tau = \mathbf{1E\text{-}7}$, and $\mathbf{L = 10}$. The BRAIN experiments use $\kappa = \mathbf{512}$, $\mathbf{m = 512}$, $\mathbf{h = 3.5}$, $\mathbf{s}_{max} = \mathbf{2048}$, $\tau = \mathbf{1E\text{-}5}$, and $\mathbf{L = 4}$. The experiment in row 33 requires 4s for skeletonization, 39s for interaction list construction, and 185s for LET exchange. All experiments are on the Maverick system with 20 OpenMP threads per node.*

effects as we increase L: the accuracy improves and the number of kernel evaluations increases. Also, note that the number of evaluations required for `ASKIT-FMM` increases much more slowly than for `ASKIT`. As L increases, more of the nodes in FMMFar are small. This in turn makes it more likely for them to be successfully merged to higher levels of the tree, thus reducing the number of kernel evaluations.

Different kernel functions. In Table 5, we show results for the Gaussian, with two different bandwidths h. The larger h (#20 and 21) requires fewer kernel evaluations but achieves greater accuracy than for $h = 0.05$ (#18 and 18). This is because the singular values of the off-diagonal blocks G decay more quickly for the larger bandwidth, and our adaptive rank skeletonization selects smaller ranks. For the Matern (#24 and 25) and polynomial (#26 and 27) kernels, we see that we can achieve small error with very few kernel interactions. The Laplace kernel is harder. Because of the singularity and the fact that we do not use geometric information, the ranks of the off-diagonal blocks are high, which results in over 20% of the direct evaluation interactions (#22). A distance-based pruning scheme will resolve this problem. Even so, the method will still be much slower than the classical FMM, which exploits analytic properties of the Laplace kernel.

Application data. In Table 6, we test three large datasets. For the selected parameters, FMM does not make as much difference here; N needs to be much bigger since as d increases, the neighbor lists overlap less. Even though the number of kernel evaluations is smaller, the time for the evaluation phase is greater for the FMM than for the treecode (see #32–37). This is because the savings from a few kernel interactions are offset by the cost of passing the skeleton potentials down the tree (Alg. 4).

Strong scaling. We report strong scaling results for `ASKIT-FMM` on the HIGGS data in Table 7. Our previous parallel implementation of `ASKIT` was limited by the excessive memory requirements for large values of κ—we only reported results for $\kappa = 1$ in [24]. However, here we are able to scale to 16K cores with $\kappa = 1024$. We see that the interaction list construction and LET exchange both scale well, while the skeletonization step does not. Note that for 16K cores, $N/(pm) = 20$ while $\log p = 10$. From our complexity bound in (16), we expect the scaling to suffer in this regime. We also point out that "Eval." is the only part of the computation that needs to be repeated for multiple right-hand sides. Although we do not report FLOPS, they can be estimated by the average evaluations per point as $2d(\#\text{K})10N/p/T_E$ (ignoring the cost of the kernel function, for example the exponential). For example, for #37 the performance is 260 GFLOPs/node and for #35 the performance is 180 GFLOPs/node.

Weak scaling. We report weak scaling results for `ASKIT-FMM` on synthetic data in Table 8. We use the uniform 64D distribution, Gaussian kernel, and 50K points per core. We use the fixed rank algorithm with $s = 512$ to enable comparisons between different runs. We see that the skeletonization phase scales linearly up to 320 cores, then increases slightly for 640 cores. As predicted in (16), we expect linear scaling until the $\log p$ term becomes significant. Since $\kappa = 128$, we see slightly worse than linear scaling in the list construction and LET exchange, since these scale as $N \log N$. The evaluation phase also scales slightly worse than linearly, corresponding to a value of ζ between zero and one (20).

Kernel regression. We demonstrate `ASKIT` on a real-world application—kernel regression and classification. Given

#cores	512	2,048	4,096	8,192	16,384
Skel. (Alg. 2)	1,295	465	370	305	269
Lists (Alg. 3)	729	177	87	46	23
LET (Eq. 7)	273	136	107	87	71
Eval. (Eq. 14)	157	67	42	28	23
Total	2,471	862	621	483	394
Efficiency	1.00	0.72	0.50	0.32	0.20

Table 7: *Strong scaling of* ASKIT-FMM *on* ***HIGGS*** *data. We report timings in seconds for different parts of the FMM algorithm.* ***"Lists"*** *includes the time to construct, merge, and invert the interaction lists.* ***"Eval"*** *includes the time to compute kernel evaluations and call Alg. 4. We use* $\kappa = \mathbf{1024}$, $\mathbf{m} = \mathbf{512}$, $\mathbf{s}_{max} = \mathbf{2048}$, $\tau = \mathbf{1E\text{-}5}$, *and* $\mathbf{L} = \mathbf{5}$. *The experiments were run on Stampede with one process per node and 16 threads per process.*

#cores	20	80	320	640
Skel. (Alg. 2)	207	207	208	214
Lists (Alg. 3)	2	2	4	5
LET (Eq. 7)	13	17	31	40
Eval. (Eq. 14)	14	17	21	23
Total	235	244	263	281
Efficiency	1.00	0.97	0.90	0.84

Table 8: *Weak scaling of* ASKIT-FMM *on* ***Uniform, 64d (6 intrinsic) data****. There are 50K points per compute core. We report timings in seconds for different parts of the FMM algorithm.* ***"Lists"*** *includes the time to construct, merge, and invert the interaction lists.* ***"Eval"*** *includes the time to compute kernel evaluations and call Alg. 4. We use the fixed rank algorithm with* $\kappa = \mathbf{128}$, $\mathbf{s} = \mathbf{512}$, *and* $\mathbf{m} = \mathbf{512}$. *The experiments were run on Maverick with one MPI process per node and 20 threads per process. In this experiment, we have fixed the bandwidth throughout to examine the scalability of he algorithm without changing other algorithm parameters. In practice, we need to adjust the bandwidth as we change the number of points.*

a training set of labeled data $\mathcal{T} = \{(x_i, y_i)\}$ and a testing set of unlabeled data $\{(x_i^{(t)})\}$, our task is to predict labels $y^{(t)}$ for the $x_i^{(t)}$. Here, we assume the labels are ± 1. For our example, we use a kernel regression classifier which chooses the label:

$$y^{(t)} = \text{sign}\left(\sum_i \mathcal{K}(x^{(t)}, x_i) w_i\right). \tag{21}$$

The model weights w_i are found by solving the linear system:

$$(\mathcal{K}_h(\mathcal{T}, \mathcal{T}) + \lambda I)\, w = y, \tag{22}$$

where $\mathcal{K}_h$ is a Gaussian kernel and λ is a cross-validation parameter. We solve (22) using a Krylov iterative method. Although K is symmetric, our approximation of it may not be. We use the GMRES method [26] and its implementation in PETSc [3]. Then, we compute the matrix vector product $\mathcal{K}(x^{(t)}, x)w$ to classify the testing set. We use ASKIT for both of these steps. We load the data and nearest neighbors, build the tree, skeletonize, exchange the LET, construct interaction lists, add the test points, and update the LET. Then in each iteration of GMRES, we use Alg. 5 to update the skeleton weights and compute the new potentials using (14).

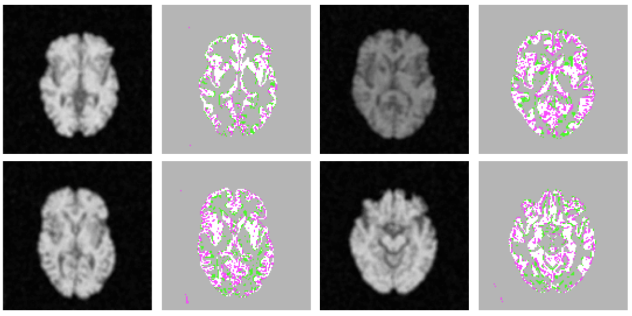


Figure 4: *We illustrate example classifications for our brain data. Each data point corresponds to a single image pixel, and the task is to identify gray matter. On the left we have the input image, and on the right we illustrate the results. A* **white** *pixel indicates correctly classified gray matter,* ***gray*** *pixels are correctly classified background,* ***green*** *is a false negative, and* ***magenta*** *is a false positive.*

#	Data	$\boldsymbol{T}_{\text{init}}$	$\boldsymbol{T}_U$	$\boldsymbol{T}_{\text{trn}}$	$\boldsymbol{T}_{\text{tst}}$	$\boldsymbol{\epsilon_2}$	Acc.
38	HIGGS	578	3	27	10	3E-4	73%
39	BRAIN	495	1	16	2	3E-7	94%

Table 9: *Results for our regression experiments.* ***"#"*** *labels the experiment,* ***"Data"*** *is the regression task,* $\mathbf{T}_{init}$ *is the setup time (tree building, skeletonization, LET, interaction list construction, and updating the LET for test points),* $\mathbf{T_U}$ *is the average time to update the skeleton weights per iteration (Alg. 5),* $\mathbf{T}_{trn}$ *is the average time to compute* Kw *per iteration (22),* $\mathbf{T}_{tst}$ *is the time to compute the test potentials (21),* $\boldsymbol{\epsilon_2}$ *is the estimated matrix approximation error for ASKIT, and* ***"Acc"*** *is the classification accuracy. We use* $h = 2.0$ *and* $\lambda = 10^{-3}$ *with 500K test points for #38 and* $h = 3.5$ *and* $\lambda = 1$ *with 819,200 test points for #39. Both experiments use* $\kappa = \mathbf{1024}$, $\mathbf{m} = \mathbf{512}$, $\mathbf{s}_{max} = \mathbf{2048}$, $\tau = \mathbf{1}E\text{-}7$, $\mathbf{L} = \mathbf{5}$, *and 100 GMRES iterations. These experiments were run on Stampede using 512 nodes, with one MPI process and 16 threads per node.*

In a real learning problem, one also needs to choose the kernel bandwidth h and parameter λ. This is done by cross-validation, which requires solving (22) for many different partitionings of the data into training and validation sets. ASKIT is well-suited to this case as well. New values of h require a new skeletonization, since the kernel function changes. However, we can evaluate the matrix-vector product in (22) for multiple values of λ without incurring the setup cost again. The time for this evaluation is T_E reported in all of our experiments.

We perform kernel-regression on an image segmentation task in medical imaging [9]. Given images of human brains from MRI tasks, we identify the portions of the image which correspond to gray matter (see Figure 4).

Regression results. We performed experiments over several values of h and λ and display the best classification results in Table 9. We highlight several features in detail. We also show illustrations from experiment #39 in Fig. 4 and the GMRES convergence history in Fig. 5. For all runs, the time per iteration is small compared to the setup time. The skeletonization and construction of the interaction lists

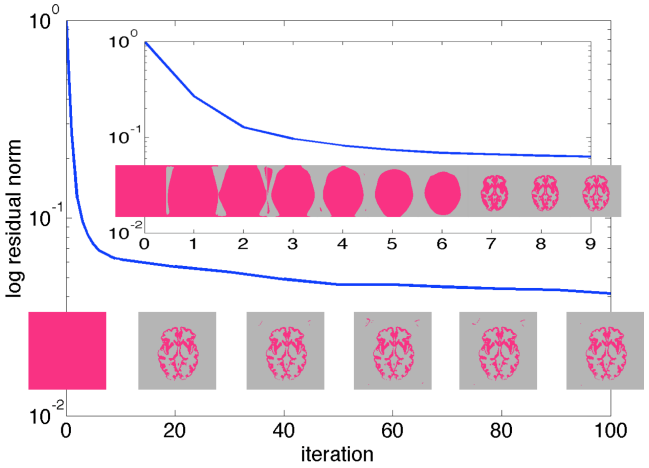


Figure 5: *We illustrate the Krylov iterates over several GMRES iterations from our experiment in #39. The images indicate the classification result over several iterations.* ***Magenta*** *pixels are classified as gray matter. The vertical axis is the logarithm of the residual norm. The inset shows the first 10 iterations in detail.*

is significant, but amortized over GMRES iterations and cross-validation for λ. This low cost per iteration is possible because `ASKIT` updates the skeleton weights efficiently (Alg. 5). Despite the fact that the BRAIN set has $d = 246$, it achieves better matrix approximation and classification accuracy with less time than for HIGGS. This is because `ASKIT` scales with d_{intr}, which is small for this set. In Figure 5 we observe that the GMRES iterations stagnate, which suggests the need for preconditioning.

The "ACC" label in Table 9 is the rate of points that are correctly classified. T_{init} contains the space partitioning tree construction, skeletonization, LET, and list building time. T_{trn} is the average training time over 100 iterations of GMRES. T_{tst} is the time to apply `ASKIT` to testing points. For the HIGGS dataset, we cross validate over $\lambda = 0.001$, 0.01, 0.1, 1, 10 and $h = 0.8$, 1.0, 1.5 and 2.0. The best parameters are $h = 2.0, \lambda = 0.001$, which gives an classification accuracy of 73% across 500,000 testing points.

We also explore parameters for the BRAIN dataset for $h = 2.5$, 3.5, 4.5 and $\lambda = 0.1, 1, 10$. The testing points consist of all pixels from 50 brain images. Figure 4 shows some segmentation results of the grey matter from MRI brain scans. The accuracy of brain with $h = 2.0, \lambda = 1.0$ is 94% across all the testing points. The Jaccard index of the grey matter is 0.60. These segmentations are not perfect due to the existence of artifacts, noise, resolution restriction of the MRI scans, and the fact that we did not solve the regression problem to higher accuracy.

5. CONCLUSION

We introduced a new FMM method for the approximation of kernel matrices in general dimensions. If the kernel function is sufficiently smooth in the far field and d_{intr} is small, `ASKIT-FMM` scales quite well with d. We outlined the main results on the complexity and convergence of the method. We reported results for several different kernels for problems on up to 246 dimensions. The evaluation phase of our FMM scheme, which is used repeatedly during cross-validation studies, scales very well.

We have only scratched the surface of FMM methods in general dimensions. Many theoretical and algorithmic issues remain open. Our skeletonization uses pivoted QR factorization, which has poor FLOP performance. Is there a way to further accelerate it? The adjustment of the different parameters to the dataset and to the accuracy requirements of the client application is now done manually. Can it be automated robustly? Combining distance-based pruning with our neighbor-pruning will improve performance significantly in lower dimensions. How can this be done efficiently?

Acknowledgements

This material is based upon work supported by AFOSR grants FA9550-12-10484 and FA9550-11-10339; by NSF grants CCF-1337393, OCI-1029022; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0010518, DE-SC0009286, and DE-FG02-08ER2585; by NIH grant 10042242; and by the Technische Universität München—Institute for Advanced Study, funded by the German Excellence Initiative (and the European Union Seventh Framework Programme under grant agreement 291763). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR, the DOE, the NIH, or the NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

6. REFERENCES

[1] A. Andoni and P. Indyk, *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, Communications of the ACM, 51 (2008), p. 117.

[2] K. Bache and M. Lichman, *UCI machine learning repository*, 2013.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[4] P. Baldi, P. Sadowski, and D. Whiteson, *Searching for exotic particles in high-energy physics with deep learning*, Nature communications, 5 (2014).

[5] J. Chen, L. Wang, and M. Anitescu, *A fast summation tree code for Matérn kernel*, SIAM Journal on Scientific Computing, 36 (2014), pp. A289–A309.

[6] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, *On the compression of low rank matrices*, SIAM Journal on Scientific Computing, 26 (2005), pp. 1389–1404.

[7] S. Dasgupta and Y. Freund, *Random projection trees and low dimensional manifolds*, in Proceedings of the 40th annual ACM symposium on Theory of computing, ACM, 2008, pp. 537–546.

[8] Z. Gimbutas and F. Rokhlin, *A generalized fast mulipole method for nonoscillatory kernels*, SIAM

Journal on Scientific Computing, 24 (2002), pp. 796–817.

[9] A. Gooya, K. M. Pohl, M. Bilello, L. Cirillo, G. Biros, E. R. Melhem, and C. Davatzikos, *GLISTR: glioma image segmentation and registration*, Medical Imaging, IEEE Transactions on, 31 (2012), pp. 1941–1954.

[10] B. Gough, *GNU scientific library reference manual*, Network Theory Ltd., 2009.

[11] A. Gray and A. Moore, *N-body problems in statistical learning*, Advances in neural information processing systems, (2001), pp. 521–527.

[12] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348.

[13] L. Greengard and J. Strain, *The fast Gauss transform*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 79–94.

[14] Greengard, L., *Fast Algorithms For Classical Physics*, Science, 265 (1994), pp. 909–914.

[15] M. Griebel and D. Wissel, *Fast approximation of the discrete Gauss transform in higher dimensions*, Journal of Scientific Computing, 55 (2013), pp. 149–172.

[16] M. Gu and S. C. Eisenstat, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM Journal on Scientific Computing, 17 (1996), pp. 848–869.

[17] T. Hofmann, B. Schölkopf, and A. J. Smola, *Kernel methods in machine learning*, The annals of statistics, (2008), pp. 1171–1220.

[18] M. Klaas, M. Briers, N. De Freitas, A. Doucet, S. Maskell, and D. Lang, *Fast particle smoothing: If I had a million particles*, in Proceedings of the 23rd international conference on Machine learning, ACM, 2006, pp. 481–488.

[19] D. Lee, P. Sao, R. Vuduc, and A. G. Gray, *A distributed kernel summation framework for general-dimension machine learning*, Statistical Analysis and Data Mining, (2013).

[20] M. W. Mahoney, *Randomized algorithms for matrices and data*, Foundations and Trends® in Machine Learning, 3 (2011), pp. 123–224.

[21] W. B. March and G. Biros, *Far-field compression for fast kernel summation methods in high dimensions*, 2014. arxiv.org/abs/1409.2802v1.

[22] W. B. March, B. Xiao, and G. Biros, *ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1089–A1110.

[23] W. B. March, B. Xiao, S. Tharakan, C. D. Yu, and G. Biros, *Robust treecode approximation for kernel machines*, in Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Sydney, Australia, 2008, pp. 1–10.

[24] W. B. March, B. Xiao, C. Yu, and G. Biros, *An algebraic parallel treecode in arbitrary dimensions*, in Proceedings of IPDPS 2015, 29th IEEE International Parallel and Distributed Processing Symposium, Hyderabad, India, May 2015.

[25] C. E. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006.

[26] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996.

[27] M. Spivak, S. K. Veerapaneni, and L. Greengard, *The fast generalized Gauss transform*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3092–3107.

[28] M. S. Warren and J. K. Salmon, *A parallel hashed octree N-body algorithm*, in Proceedings of Supercomputing, The SCxy Conference series, Portland, Oregon, November 1993, ACM/IEEE.

[29] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference*, Springer Science & Business Media, 2004.

[30] H. Wendland, *Scattered data approximation*, vol. 17, Cambridge university press, 2004.

[31] C. Williams and M. Seeger, *Using the Nyström method to speed up kernel machines*, in Proceedings of the 14th Annual Conference on Neural Information Processing Systems, 2001, pp. 682–688.

[32] B. Xiao, *Parallel algorithms for the generalized n-body problem in high dimensions and their applications for bayesian inference and image analysis*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 8 2014.

[33] B. Xiao and G. Biros, *Parallel algorithms for nearest neighbor searches*, 2015. padas.ices.utexas.edu/static/papers/knn.pdf.

[34] L. Ying, G. Biros, and D. Zorin, *A kernel-independent adaptive fast multipole method in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.

[35] C. D. Yu and G. Biros, *Optimized direct kernel summation on x86 architectures*, 2015. in preparation.