**Abstract**—Kernel matrices appear in machine learning and non-parametric statistics. Given $N$ points in $d$ dimensions and a kernel function that requires $O(d)$ work to evaluate, we present an $O(dN \log N)$-work algorithm for the approximate factorization of a regularized kernel matrix, a common computational bottleneck in the training phase of a learning task. With this factorization, solving a linear system with a kernel matrix can be done with $O(N\log N)$ work. Our algorithm only requires kernel evaluations and does not require that the kernel matrix admits an efficient global low rank approximation. Instead our factorization only assumes low-rank properties for the off-diagonal blocks under an appropriate row and column ordering. We also present a hybrid method that, when the factorization is prohibitively expensive, combines a partial factorization with iterative methods. As a highlight, we are able to approximately factorize a dense $11M \times 11M$-kernel matrix in 2 minutes on 3,072 x86 “Haswell” cores and a $4.5M \times 4.5M$ matrix in 1 minute using 4,352 “Knights Landing” cores.

**I. INTRODUCTION**

Let $\mathcal{X}$ be a set of $N$ points $x \in \mathbb{R}^d$ and let $K(x_i, x_j) : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ be a given kernel function. The kernel matrix is the $N \times N$ matrix whose entries are given by $K_{ij} = K(x_i, x_j)$ for $i, j = 1, \ldots, N$, $x_i, x_j, \in \mathcal{X}$.

Kernel matrices appear in unsupervised and supervised statistical learning, Gaussian process regression, and non-parametric statistics [8], [12], [27], [33]. Solving linear systems with kernel matrices is an algebraic operation that is required in many kernel methods. The simplest example is ridge regression in which we solve systems with kernel matrices is an algebraic operation that is required in many kernel methods. The simplest example is ridge regression in which we assume that $K$ is typically large $N$ because $K$ is typically dense. For example, consider the Gaussian kernel,

$$K(x_i, x_j) = \exp \left( -\frac{1}{2h^2} \|x_i - x_j\|^2 \right),$$

where $h$ is the kernel bandwidth. For small $h$, $K$ approaches the identity matrix whereas for large $h$, $K$ approaches the rank-one constant matrix. The first regime suggests sparse approximations while the second regime suggests global low-rank approximations. But for the majority of $h$ values, $K$ is neither sparse nor globally low-rank. Direct factorization of $\lambda I + K$ requires $O(N^3)$ work, whereas a Krylov iterative method costs $O(N^2)$ work per iteration and may require 1000s of iterations. This complexity barrier has limited the use of kernel methods for large-scale problems [6], [29].

**Contributions.** We exploit hierarchically low-rank approximations in which we assume that $K$ can be approximated well by $D + UV$, where $D$ is block-diagonal and the $U$ and $V$ matrices have low rank. Using such a decomposition, we improve the factorization algorithm presented in [36]. In that paper the block-diagonal plus sparse decomposition was done using ASKIT [1] a method introduced in [21], [22] (see §II-A). ASKIT approximates $K$ in $O(dN \log N)$ time and the algorithm in [36] factorizes the ASKIT approximation in $O(N \log^2 N)$ time (see §II-B). Roughly speaking, ASKIT is based on the approximation of $K$ as the sum of a block-diagonal matrix and a low-rank matrix followed by recursion for each diagonal block. We refer this process as the construction of the hierarchical representation of $K$. Once we have this representation, we can factorize $K$ by applying recursively the Sherman-Morrison-Woodbury (SMW) formula. The factorization has to be done for different values of $\lambda$ during cross-validation studies. Therefore optimizing the factorization is crucial for the overall performance of a kernel method. In this paper, we extend the factorization scheme presented in [36] in several ways:

- We present an algorithm that factors the ASKIT approximation in $O(N \log N)$ time instead of $O(N \log^2 N)$ and we demonstrate its performance on several datasets (see §II-C and §V).

- We present a hybrid level-restricted factorization scheme that reduces dramatically the factorization time and we demonstrate its performance on several datasets (see §II-C). The new method can be used with matrices for which [36] fails.

- We study performance on Intel’s “Knights Landing” (KNL) architecture. We introduce an optimized matrix-free kernel summation that reduces the storage requirements of the factorization without having a very significant impact on wall-clock time (see §II-D).

In our numerical experiments, we measure the performance of the method on several different datasets. ASKIT has been applied to polynomial, Matern, Laplacian, and Gaussian
kernels in arbitrary dimensions. Due to space limitations we only present the factorization results for the Gaussian kernel function. Also the Gaussian kernel is among the hardest to compress in high dimensions. We examine the performance of the method for different bandwidth ranges that are relevant to learning tasks, its sensitivity to the regularization parameter $\lambda$, its performance as we increase the number of points, and its numerical stability (see [36]).

Limitations. Not all kernel matrices admit a good hierarchical low-rank decomposition. Typically this is related to the intrinsic dimensionality of the dataset at different scales. So ASKIT and subsequently our method can fail. If ASKIT can compress the matrix, the second potential point of failure is the choice of regularization parameter. If it is too small, our algorithm (as well as [36]) can become numerically unstable. We can numerically detect the instability, but it is not clear how to fix it while maintaining the log-linear complexity of the algorithm. However, small regularization often results in poor learning performance so this corner case is not important in applications. We discuss this in more detail in [III] and [V] Also our methods cannot be applied to hierarchical decompositions in which $D$ is sparse and not just block diagonal. For such decompositions, our method can be used as a preconditioner, as discussed in [36].

Related work. Nystrom methods and their variants [7], [13], [28], [34] can be used to build fast factorizations. However, not all kernel matrices can be approximated well by Nystrom methods [15], [18], [23], [31]. Factorization methods based on hierarchical decomposition have been studied for kernel matrices from points in two or three dimensions [1], [2], [4], [9], but less so in high dimensions with a few exceptions [15], [36]. Early works discussing parallel operations for hierarchical matrices on shared memory systems include bulk synchronous parallelization [16] and DAG-based task parallelism [17]. Distributed factorization and operations were discussed in [14], [32]. The difficulties of generalizing low-dimensional factorizations in high-dimensions are discussed in [21], [36].

II. METHODS

We begin with a sketch of hierarchical matrices and direct solvers in [II-A]. We also briefly summarize the ASKIT algorithm which we use as the basis for our new methods. We describe parallel factorization schemes in [II-B] and highlight the novelty of our approach over [36]. We then introduce our hybrid iterative/direct solver in [II-C].

A. Hierarchical Matrices and Tree Codes

Broadly speaking, we consider a matrix $K \in \mathbb{R}^{N \times N}$ to be hierarchical if it can be partitioned as

$$K = \begin{bmatrix} K_{11} & K_{1r} \\ K_{r1} & K_{rr} \end{bmatrix} = \begin{bmatrix} K_{11} & 0 \\ 0 & K_{rr} \end{bmatrix} + \begin{bmatrix} 0 & K_{1r} \\ K_{r1} & 0 \end{bmatrix}.$$  (2)

where the off-diagonal blocks $K_{1r}$ and $K_{r1}$ can be accurately approximated by a low-rank factorization and the on-diagonal blocks $K_{11}$ and $K_{rr}$ are themselves hierarchical. Note that the low-rank structure is not invariant on permutations, it very strongly depends on the ordering of the columns (or rows since the matrix is symmetric). For notational convenience we write $K \approx \tilde{K} = D + UV$, where $U$ and $V$ are rank $s$ and $D$ is also hierarchical, where use $\tilde{K}$ to indicate the approximate kernel matrix.

Inverting hierarchical matrices. When $K$ admits this hierarchical low-rank approximation, then we can efficiently approximate $\tilde{K}^{-1}$ using the Sherman-Morrison-Woodbury formula along with recursion:

$$\tilde{K}^{-1} = (I + VW)^{-1}D^{-1} = (I - W(I + VW)^{-1}V)D^{-1} = (I - WZ^{-1}V)D^{-1},$$  (3)

Recursion is used to invert $D^{-1}$. After obtaining $D^{-1}$ we compute $W = D^{-1}U$ for a rank-$s$ matrix $U$ and factorize the smaller reduced system $Z = I + VW$. The scheme can be easily extended to invert $M + K$.

To turn this formulation into an algorithm, we need (1) a method to partition $K$ so that off-diagonal blocks have low-rank, (2) an efficient way to compute the low-rank factors $U$ and $V$, and (3) a scheme to construct the inverse. For the first two tasks we use ASKIT, a method we recently developed [21], [23]. ASKIT uses geometric information (the input points) to permute $K$ by partitioning the points recursively using a binary tree. Interactions between points in a treenode correspond to diagonal blocks of $K$. In the recursion, the children of the node can be used to define the block partitioning of the parent block, similar to [2]. Next, we summarize ASKIT features that are necessary for this paper. Please see [23] for the complete details on ASKIT.

Partitioning the matrix. We use a ball tree [26] to partition $\mathcal{X}$. Starting with the root node (which contains the entire data set), nodes are partitioned into two children (with an equal number of points) by a splitting hyperplane. This recursive splitting terminates when a node has less than $m$ points, a user-specified parameter. The root has level $l = 0$ and the leaves $l = D = \log_2(N/m)$, the depth of the tree. In the following, we overload $\alpha$, $\beta$ to indicate both binary tree nodes and the indices of the points that belong to these nodes; $|\alpha|$ is the number of points in $\alpha$; and 1, $r$ indicate the left and right children of the node. We define $\mathcal{X} \in \mathbb{R}^{d \times N}$ to be the matrix of all points and $\mathcal{X}_\alpha$ to be the points owned by tree node $\alpha$, i.e., $\mathcal{X}_\alpha = \{x_i | \forall i \in \alpha \}$. The points in a leaf node $\alpha$ are unordered.

Computing low rank approximations. Let $\alpha$ be the points in a leaf node. Let $\mathcal{S} = \{1, \ldots, N\} \setminus \alpha$. The skeletonization of a node $\alpha$ is a rank-$s$ approximation of $K_{\mathcal{S} \alpha}$ using $s$ columns of $K_{\mathcal{S} \alpha}$. We refer to these columns as the skeleton of $\alpha$, denoted by $\tilde{\alpha}$. Skeletonization is done using the Interpolative Decomposition (ID) [11]. Using a pivoted rank-revealing QR factorization, the ID finds $\tilde{\alpha}$ and $P_{\tilde{\alpha}} \in \mathbb{R}^{s \times |\alpha|}$ such that

$$K_{\mathcal{S} \alpha} \approx K_{\mathcal{S} \tilde{\alpha}} P_{\tilde{\alpha}}.$$

(4)
Algorithm II.1 \([\bar{\alpha}, P_\bar{\alpha}] = \text{Skeletonize}(\alpha)\)

if \(\alpha\) is leaf then return \([\bar{\alpha}, P_{\bar{\alpha}}] = \ID(\alpha)\);
\([\bar{1}, \bar{r}] = \text{Skeletonize}(1)\); \([\bar{r}] = \text{Skeletonize}(r)\);
return \([\bar{\alpha}, P_{\bar{\alpha}}[\bar{r}]] = \ID(\{\bar{1}, \bar{r}\})\);

The first \(s\) pivots from the QR define \(\bar{\alpha}\). Using the QR, we can compute \(P_{\alpha_{\bar{\alpha}}} = K_{\bar{\alpha}\bar{\alpha}}^tK_{\bar{\alpha}\alpha}\). This scheme however results in \(O(dN^2m)\) complexity for the overall factorization. We can turn it to a \(O(d \log N m)\) scheme by sampling a small subset \(S'\) of \(S\) and using it instead of \(S\). The approximation rank \(s\) is chosen such that \(\sigma_{s+1}(K_{S'/\alpha})/\sigma_1(K_{S'/\alpha}) < \tau\), where \(\tau\) is user-specified and \(\sigma\) are the singular values estimated by the diagonal of the rank-revealing QR.

For a non-leaf \(\alpha\), we first compute the skeletons \(\bar{1}\) and \(\bar{r}\) of the children of \(\alpha\) and then we compute the skeleton \(\bar{\alpha} \in \bar{1} \cup \bar{r}\) and the projection matrix \(P_{\alpha[\bar{r}]}\) using another ID decomposition (Algorithm II.1). Once the skeletonization of every node (but the root) is computed, we can compute the ID decomposition (Algorithm II.1). Once the skeletonization of every node (but the root) is computed, we can compute the ID decomposition (Algorithm II.1).

We factorize \(\tilde{\lambda}_{\bar{\alpha}}\), which in turn requires traversing all the descendants of \(\alpha\) (the subtree rooted at \(\alpha\)) and recursively applying (7). (We introduced this scheme in [36] and results in \(O(dN \log^2 N)\) complexity.) But as we will see shortly this subtree traversal is not necessary. Once we have \(W_{\alpha}\), we use the SMW formula to invert \((I + W_{\alpha}V_{\alpha})^{-1}\). This inverse requires \(W_{\alpha}Z_{\alpha}^{-1}V_{\alpha}\), which in terms of the block decomposition of \(\alpha\), can be written as

\[
P_{\bar{\alpha}}^{-1}[\bar{r}] = [\tilde{K}_{\bar{r}} \tilde{K}_{\bar{r}r}^{-1} \tilde{K}_{\bar{r}r}]^{-1} [\tilde{K}_{\bar{r}} \tilde{K}_{\bar{r}r}] P_{\bar{\alpha}[\bar{r}]}^{-1} \approx [\tilde{K}_{\bar{r}} \tilde{K}_{\bar{r}r}^{-1} \tilde{K}_{\bar{r}r}]^{-1} [\tilde{K}_{\bar{r}} \tilde{K}_{\bar{r}r}] P_{\bar{\alpha}[\bar{r}]}^{-1} \tag{8}\]

Since \(\alpha\) is the parent of \(1\) and \(r\), \(P_{\bar{\alpha}[\bar{r}]}^{-1}\) and \(P_{\bar{\alpha}[\bar{r}]r}\) have been already computed. \(K_{\bar{\alpha}[\bar{r}]r}\) and \(K_{\bar{\alpha}[\bar{r}]1}\) are computed by GEMM, and the reduced system is factorized by GETRF.

We can exploit a “telescoping” relation between \(P_{\bar{\alpha}[\bar{r}]r}\) and \(P_{\bar{\alpha}[\bar{r}]1}\). We say that \(P_{\bar{\alpha}[\bar{r}]}\) is “telescoped” from \(P_{\bar{\alpha}[\bar{r}]1, \bar{r}}\) and \(P_{\bar{\alpha}[\bar{r}]1}\) because is computed by formula in the box below.

\[
P_{\bar{\alpha}[\bar{r}]r}^{-1}\approx [\tilde{K}_{\bar{r}r}^{-1} \tilde{K}_{\bar{r}r}]^{-1} [\tilde{K}_{\bar{r}r}^{-1} \tilde{K}_{\bar{r}r}] P_{\bar{\alpha}[\bar{r}]}^{-1} \tag{9}\]

The calculation in the box requires just GEMM operations from the children (1 and \(r\)) but not all descendants. Since \(P_{\bar{\alpha}[\bar{r}]} = K_{\bar{2}r}^{-1}P_{\bar{\alpha}[\bar{r}]} = (I + W_{\alpha}V_{\alpha})^{-1}P_{\bar{\alpha}[\bar{r}]1}\), we can replace \(D_{\alpha}^{-1}P_{\alpha[\bar{r}]1}\) with (9). Now we find that \(P_{\bar{\alpha}[\bar{r}]r}\) can also be telescoped by \(P_{\bar{\alpha}[\bar{r}]1}\) and \(P_{\bar{\alpha}[\bar{r}]1}\) as

\[
P_{\bar{\alpha}[\bar{r}]} = (I + [\tilde{P}_{\bar{1}r} \tilde{P}_{\bar{r}r}] V_{\alpha})^{-1} [\tilde{P}_{\bar{1}r} \tilde{P}_{\bar{r}r}] P_{\bar{\alpha}[\bar{r}]} \tag{10}\]

Notice that we no longer need to solve \(\tilde{K}_{\bar{r}r}^{-1}\) and \(\tilde{K}_{\bar{r}r}\) in (10). Thus, no tree traversal is required. In the leaf level (base case), \(P_{\bar{\alpha}[\bar{r}]1}\) is computed directly from \(K_{\bar{2}r}^{-1}P_{\bar{\alpha}[\bar{r}]1}\).

Given these formulas and the skeletonization computed in Algorithm II.1, we compute the factors needed for the direct solver in a postorder traversal of the tree (Algorithm II.2). If \(\alpha\) is a leaf node, we factorize \(\mathcal{L} + K_{\bar{\alpha}\alpha}\) using an LU factorization. Otherwise, we compute \(K_{\bar{1}\alpha}\) and \(K_{\bar{r}\alpha}\). Notice that \(P_{\bar{1}\alpha}\) and \(P_{\bar{r}\alpha}\) are computed in the previous recursion; thus, we can form and factorize the reduced system \(Z_{\alpha}\). Finally, \(P_{\bar{\alpha}\alpha}\) is telescoped using (10), thus \(\text{solve}(\alpha, W_{\alpha} P_{\bar{\alpha}[\bar{r}]\alpha}, \text{false})\) (Algorithm II.3) will not invoke recursion.

This algorithm improves on the one in [36] by removing the extra subtree traversals that result in \(O(N \log^2 N)\) complexity. Instead, our algorithm exploits the nested structure of \(P_{\bar{\alpha}\alpha}\) resulting in an \(N \log^2 N\) complexity for the factorization. In some of our largest runs, this resulted in over 3× speedup without any change in the accuracy.
For example, the yellow process owns in the direct solver construction and show which process owns which factor. Each process

\[
\tilde{K}_\alpha = \text{Solve}(\alpha, W_\alpha P_{[\alpha]} \tilde{\alpha}, \text{false}) \text{ using (10)}. 
\]

Algorithm II.2 Factorize(\alpha)

if \( \alpha \) is leaf then

LU factorization \( \lambda I + K_{\alpha a} \).

\( W_\alpha = P_{\tilde{\alpha} \alpha} \) and \( P_{[\tilde{\alpha} \alpha]} = I \).

else

Factorize(1) and Factorize(\beta).

Form \( W_\alpha \) with \( \tilde{P}_{[1]} \), \( \tilde{P}_{[\beta]} \), and \( V_\alpha \) with \( K_{\alpha} \), \( K_{[\beta]} \).

LU factorize the reduced system \( Z_\alpha \) in (8).

\( \tilde{P}_{[\tilde{\alpha} \beta]} = \text{Solve}(\alpha, W_\alpha P_{[\beta]} \tilde{\beta} \tilde{\alpha}, \text{false}) \) using (10).

Algorithm II.3 \( w = \text{Solve}(\alpha, y, \text{do recur}) \)

if \( \alpha \) is leaf then LU solver \( w = (\lambda I + K_{\alpha a})^{-1} y \)

else

if \( \text{do recur} \) then \( v = [\text{Solve}(1, y_1, \text{true}); \text{Solve}(\beta, y_2, \text{true})] \).

Compute \( w = u - W_\alpha Z_\alpha^{-1} V_\alpha u \) using (5).

\[ \begin{align*}
\text{Figure 1: The top four levels of the tree and the corresponding blocks of the matrix } \tilde{K}.
\text{The nodes belonging to each process are highlighted in a single color. Each process factorizes its own portion of the tree independently. We also highlight the factors used in the direct solver construction and show which process owns which factor. Each process owns a diagonal block and all factors in the same column and the same row. For example, the yellow process owns } P_{[\alpha]} \text{ and } K_{\alpha \beta} \text{ at level 1; similarly it owns } P_{[\beta]} \text{ and } K_{[\beta]} \text{ at level 0.}
\end{align*} \]

We then describe how to apply \( \tilde{K}_{\tilde{\alpha} \alpha} \) to a vector \( y \), shown in Algorithm II.3. If \( \alpha \) is a leaf node, we can directly invoke an LU solver to obtain \( w = K_{\alpha \alpha}^{-1} y \). Otherwise, we have two situations. If Algorithm II.3 is called by Factorize (do_rec is false), then we know \( u = W_\alpha P_{[\tilde{\alpha} \alpha]} \).

Thus, no recursion is required. While Solve is called to solve a random \( u \), then we need to solve \( K_{[\beta]}^{-1} \) \( y_1 \) and \( K_{[\beta]}^{-1} y_2 \) recursively and compute (3) with GEMV on \( \alpha \) and \( \alpha \) and an LU solve \( \text{GETRS} \) on \( Z_\alpha \).

Therefore, the complete algorithm consists of constructing the tree, calling Algorithm II.1 then Algorithm II.2 and Algorithm II.3 each called on the root of the tree.

Parallel direct solver. The parallelization is essentially identical to the scheme proposed in (36). Each subtree (a set of points \{x\}) is assigned to a distributed-memory process (or a worker). Although we described recursive version of our algorithms we use level-by-level traversals combined with shared or distributed memory parallelism (depending on the level) across nodes in the same level. If the number of nodes is less than the number of physical cores, the OpenMP

Algorithm II.4 DistFactorize(\alpha, q)

if \( \alpha \) is at level log \( p \) then Factorize(\alpha).

else

DistFactorize(c, \( q/2 \)).

\( \{i < \frac{q}{2}\} \quad \{i \geq \frac{q}{2}\} \)

0. Send \( \hat{I} \).

(\( q \)) Recv, Bcast \( \hat{I} \).

(\( q \)) Recv, Bcast \( \hat{F} \).

Reduce \( K_{\tilde{\alpha}}(x) \hat{P}(x) \hat{I} \).

Reduce \( K_{\tilde{\alpha}}(x) \hat{P}(x) \hat{F} \).

0. Send \( \hat{V} \).

(\( q \)) Recv \( \hat{V} \).

(\( q \)) Bcast \( \hat{V} \).

(\( q \)) Send \( \hat{Z} \).

(\( q \)) Bcast \( \hat{Z} \).

\( w = u - \hat{P}(x) \hat{V} \).

\( w = u - \hat{P}(x) \hat{V} \).

Algorithm II.5 \( w = \text{DistSolve}(\alpha, y, q, \text{do recur}) \)

if \( \alpha \) is at level log \( p \) then \( w = \text{Solve}(\alpha, y, \text{do recur}) \).

else

if \( \text{do recur} \) then \( u = \text{DistSolve}(c, y, \frac{q}{2}, \text{do recur}) \).

\( \{i < \frac{q}{2}\} \quad \{i \geq \frac{q}{2}\} \)

Reduce \( K_{\tilde{\alpha}}(x) u \).

Reduce \( u \).

0. Recv \( u \).

(\( q \)) Send \( u \).

(\( q \)) Send \( u \).

0. Bcast \( u \).

(\( q \)) Recv \( u \).

(\( q \)) Bcast \( u \).

\( w = u - \hat{P}(x) \hat{P} \).

\( w = u - \hat{P}(x) \hat{P} \).

nested construct is enabled such that each thread will invoke parallel BLAS or LAPACK routines. If we have \( p \) processes, then above level log \( p \) of the tree, we have to communicate to compute factors, since the terms needed are distributed among processes. We use the Message Passing Interface (MPI) library for distributed memory communication.

In Figure 1 we summarize the distributed-memory algorithm. The four colors represent four different MPI ranks and the nodes they own. The tree on the right shows \( l = 2 \) treenodes are uniquely assigned to ranks, but treenodes with \( l > 2 \) are shared among ranks. To facilitate collective communication, each distributed treenode creates a local communicator, which equally divides the ranks of the parent. We use \( \{i\} \) to denote the \( i_{th} \) MPI rank in the local communicator. Consider the communicator of \( \alpha \), which involves \( q \) ranks. Let \( c \) denote the child of \( \alpha \) that \( \{i\} \) owns. Then \( c = 1 \) if \( \{i < \frac{q}{2}\} \). Otherwise, \( c = \{\beta\} \) for a distributed node \( \alpha \), data points \( \{x\} \), owned by \( \{i\} \) are never required by other MPI processes, \( X_\alpha = X_1 \cup X_\beta = (\cup \{i < \frac{q}{2}\} \{x\}) \cup (\cup \{i \geq \frac{q}{2}\} \{x\}) \).

However, skeletons \( \hat{\alpha} \) and \( P_{[\hat{\alpha}]} \) are only stored in \( \{0\} \). When its sibling needs this information, we exchange the information using a SendRecv between \( \{0\} \) and \( \{\frac{q}{2}\} \) using the parent communicator of \( \alpha \) and its sibling. Once received, \( \alpha \) and the sibling communicator can Bcast to every processes in their groups.

Algorithm II.4 describes the recursive distributed factorization. In each node \( \alpha \), ranks \( \{i < \frac{q}{2}\} \) requires skele-
Algorithm II.8

DistFactorize

1. The basic idea here is not to store and factorize any blocks of the system that are owned by processes other than the root.

2. Non-structured factors are distributed among processes. Once reaching level $l$, if we are not at the root process, we broadcast $P_{\beta\beta}K_{\beta\beta}$ and $P_{r\beta}K_{r\beta}$ to the root process.

3. At level $L$, the root process solves $P_{\beta\beta}K_{\beta\beta}$ and $P_{r\beta}K_{r\beta}$,

4. The output is again broadcast back to all other processes.

5. This requires reduction and sends $K_{\beta\beta}$ and $K_{r\beta}$ to the root process.

6. The root process factorizes $K_{\beta\beta}$ and $K_{r\beta}$ sequentially. The output is then broadcast back to all other processes.

7. This process continues until all processes are at the root process.

8. The iterative solver requires the ability to compute the reduced system $L_2^{-1}w_2$.

9. The reduced system is solved iteratively by GMRES. Algorithm II.9

10. Algorithm II.9

11. Algorithm II.6

12. Algorithm II.8

13. Algorithm II.6

14. Algorithm II.9

15. Algorithm II.6

16. Algorithm II.8

17. Algorithm II.9

18. Algorithm II.6

19. Algorithm II.8

20. Algorithm II.9

21. Algorithm II.6

22. Algorithm II.8

23. Algorithm II.9

24. Algorithm II.6

C. Fast hybrid solver

As discussed level-restriction is necessary when an off-diagonal block is no longer low-rank. We refer to the set of nodes that are skeletonized but whose parent did not as the skeletonization frontier $A$. In Figure 2 the yellow nodes define the frontier. Nodes “above” (i.e., closer to the root) the frontier cannot be skeletonized.

In this case the SMW formulation can still be used but $D$, $U$, and $V$ have as many blocks as the number of nodes above $A$. Therefore, the $Z$ matrix will be quite large. For example, if the frontier $A$ consists of all the nodes at $L$, and a node in $A$ has $s$ skeletons, then the size of $Z$ will be $2^L s$. If we compute the full factorization, the cost will be $O(2^{2L+1}s^2N + 2^{3L}s^3)$ in work and $O(2^{2L}sN)$ in storage. The basic idea here is not to store and factorize any $Z$, $W$, and $V$ factors for those unskeletonized nodes, but instead use a matrix-free Krylov method. We refer to this approach as the “hybrid method”, since we factorize only up to frontier.

Level restriction reduces the system that needs to be solved from $N$ to $2^L s$.

Partial factorization. We still factorize skeletonized nodes bottom up until reaching the frontier $A$. In Figure 2 these treenodes (yellow and khaki) we factorize are diagonal blocks $D$ (yellow) on the left. Then, conceptually, we coalesce all $W$ (blue) and $V$ (green) factors for nodes above $A$. Notice that we can still apply SMW on this partial factorization in the form of Figure 3. Rather than continuing to factorize these unskeletonized nodes, we switch to an iterative solver for the reduced system $(I + VW)^{-1} w$.

In Algorithm II.6 we show this hybrid algorithm. $D^{-1}$ is computed by Algorithm II.5 on those skeletonized nodes. The iterative solver requires the ability to compute the MatVec for $W$ and $V$. Algorithm II.7 (MatVecW) computes $K_{\beta\beta} K_{\beta\beta}$ for all nodes above (including) the frontier. In the distributed tree, MatVecW performs a reduction on $\{x\}$.
thus, an AllReduce is required at the end such that all MPI ranks get the same output. On the other hand, MatVecW is supposed to perform a scattering on \( K \) and stored or they can be used in a matrix-free manner, by multiplying submatrices of \( K \) in the solving phase can be done in \( O(sN \log N) \) work. Using the complexity above, we derive \( T^F(N) = 2T^s(N/2) + O(Ns^2 + s^3) = O(s^2N \log N) \). Notice that solving \( \hat{P}_{\alpha \tilde{\alpha}} = \hat{K}_{\alpha \tilde{\alpha}}^{-1} \) does not require traversing to the leaf level. Instead \( \hat{P}_{\alpha \tilde{\alpha}} \) only takes \( O(N) \) work.

### III. Theory

Here, we present some theoretical complexity guarantees and discuss the stability of our direct solver.

**Work.** We present the complexity analysis of Algorithms [II.2] [II.3] and [II.6]. Throughout, we fix the leaf size \( m \), level restriction \( L \), and maximum skeleton size \( s \). \( T^s(N) \) denotes the complexity of Algorithm [II.2] and \( T^F(N) \) of Algorithm [II.3] each for \( N \) points. Since Solve does either an LU solve or matrix-vector multiply in each step, we have

\[
T^s(N) = 2T^s(N/2) + O(Ns^2 + s^3) = O(s^2N \log N). \tag{12}
\]

In the hybrid solver, each \( (I + VW)x \) operation requires \( O(2^LsN) \) work. To summarize, both Algorithm [II.2] and Algorithm [II.3] take \( O(N \log N) \) work, and Algorithm [II.6] takes \( O(N^2 \log N) \) with additional \( O(N) \) for each iteration if \( L \) is independent from \( N \).

**Communication.** The communication cost for the solving phase is \( O(s \log p) \) per level. To traverse the whole tree, \( O(s^2 \log p) \) is required per right hand side. However, during the factorization the solving phase does not recur. Thus, instead of \( O(s^2 \log p) \) for \( s \) right hand sides, there is only \( O(s^2 \log p) \) communication per level. Overall, the communication cost for the full factorization is \( O(s^2 \log p) \) since there are \( \log p \) distributed levels.

**Memory.** The memory cost of our methods depend on level restriction \( L \) and maximum skeleton size \( s \). These requirements are in addition to the cost to store the coordinates and skeleton information for ASKIT, reported in [21]. In our direct solver, we require the factors \( U, V, I + VW \) for each level of the tree below the level-restriction \( L \) in which skeletonization stops. This requires \( O(2sN + s^2) \) per level. Therefore, the overall memory required for our method is

\[
O \left( 2sN + s^2 \right) \left( \log \left( \frac{N}{m} \right) - L \right). \tag{14}
\]

Using GSKS can reduce \( sN \log(N/m) \) to \( O(1) \) by computing \( V \) on the fly. Recomputing \( W \) with \( \hat{P}_{\alpha \tilde{\alpha}} \) can reduce another \( sN \log(N/m) \) to \( sN \). Using both schemes

### Table I

| Arch | \( d \) | \( 4 \) | \( 20 \) | \( 68 \) | \( 132 \) | \( 260 \) |
|------|------|------|------|------|------|------|
| Haswell | MKL+VNL | 31 | 53 | 72 | 115 | 305 |
| 16K | GSKS | 321 | 465 | 512 | 634 | 687 | 680 |
| 16K | MKL+VNL | 12 | 93 | 132 | 416 | 636 | 916 |
| Haswell | GSKS | 703 | 888 | 1067 | 1246 | 1334 | 1449 |
| GSKS | 20132 | 31 | 53 | 72 | 115 | 190 | 305 |
| 321 | 465 | 512 | 634 | 687 | 680 | 479 | 888 | 903 | 975 | 1220 | 1345 |
| MKL | 479 | 888 | 903 | 975 | 1220 | 1345 | 16K | GSKS | 30 | 52 | 70 | 110 | 180 | 284 |
| 8K | GSKS | 479 | 888 | 903 | 975 | 1220 | 1345 | 16K | GSKS | 250 | 359 | 384 | 420 | 477 | 468 |
| Haswell | GSKS | 11 | 56 | 76 | 116 | 370 | 578 |
| 4K | GSKS | 341 | 445 | 464 | 510 | 858 | 1015 |

The reference implementation uses MKL _DGEMM_ and VML _VEXP_.

The Gaussian kernel summation efficiency of \( 16K \times 16K \times d, 8K \times 8K \times d, \) and \( 4K \times 4K \times d \) in GFLOPS. GSKS can be found in https://github.com/ChenhanYu/ks. The

GSKS fuses \( K \) (kernel function) and \( \text{GEMV} \) (reduction) into \( \text{GEMM} \) (semi-ring rank-\( d \) update). [35] uses the same idea to fuse nearest-neighbor search into \( \text{GEMM} \). With a BLIS-like framework [30], matrix-matrix multiplication (\( C = AB \)) is divided into subproblems. A small subproblem that fits \( C \) into registers is implemented in vectorized assembly or intrinsic to maximize FLOPS throughput. The idea is to directly perform kernel evaluation and the \( \text{GEMV} \) on \( C \) while it is still in the register and only store back a vector \( w \). In short, for a typical kernel summation that involves an \( m \times n \times d \) \( \text{GEMM} \) with \( O(md + nd + mn) \) MOPS (Memory Operations) in the best known method, GSKS can achieve \( O(mnd) \) FLOPS but with only \( O(md + nd) \) MOPS.

This helps the computation become less memory bound even with small \( d \). In this work, we implement this idea in AVX2 and AVX512 for Haswell and KNL architectures. We present the performance of these two different approaches in Table I. Due to the \( O(mn) \) memory saving, GSKS is about \( 3 \sim 30 \times \) faster than the best known method on KNL for large problem size \( \mathbb{F} \) and \( d < 68 \). We see that using GSKS significantly outperforms using the standard approach.
yields $O(s^2(\log(N/m) - L) + sN)$ storage with $O((d + s^2)N \log N)$ work (still $O(N \log N)$ asymptotically).

**Stability.** Overall, the stability of our method is related to the conditioning of $(\lambda I + \tilde{K})$, $D$ and the reduced system $(I + VW)$. We use $\kappa = \sigma_1/\sigma_{\max}$ to denote the 2-norm condition number of a matrix where $\sigma_1$ and $\sigma_{\max}$ are the largest and smallest singular values of the matrix. [10] suggests that when either $U$ or $V$ are orthonormal, then $\kappa(I + VW) \leq \kappa(D)\kappa(\tilde{K})$. Although, our $U$ and $V$ are not orthonormal, in our experience $\kappa(I + VW)$ does not have a conditioning problem when $D$ and $K$ are well-conditioned. The relation between $\kappa(\lambda I + D)$ and $\kappa(\lambda I + K)$ is more interesting. In general when $h$ shrinks, we expect $K$ to become more diagonally dominant and thus better conditioned. However, counter to this intuition, it is possible for $D$ to become more poorly conditioned as $h$ shrinks.

Since $D$ is a submatrix of $K$, we have $\sigma_1(D) \leq \sigma_1(K)$ and $\sigma_n(D) \leq \sigma_n(K)$. When $\sigma_n(K) < \lambda$, then $\kappa(\lambda I + D) < \kappa(\lambda I + K)$ since $\lambda$ dominates in the denominator. However when $\sigma_n(K) > \lambda$, $\kappa(\lambda I + D)$ can grow even as $\kappa(\lambda I + K)$ remains small. If this case happens in many levels of our factorization, then the method is not stable. With narrow bandwidths where $K$ approaches a (blocked)-diagonal matrix, $\sigma_n > \lambda$ may occur.

Under the framework of hierarchical matrices, the pivoting rows we can choose during the $D$ factorization are limited to the skeleton rows. Thus, even $\kappa(\lambda I + K)$ is not bad, $(\lambda I + D)$ can be unstable due to the aggressive pivoting strategy if $\lambda$ is small. Our methods can detect this situation, but avoiding this case entirely (or fixing it) is not straightforward.

### IV. Experimental Setup

We performed numerical experiments on Haswell and KNL architectures with four different setups to examine the accuracy and efficiency of our methods. Especially, we want to demonstrate (1) the complexity improvement against [30], (2) FLOPS efficiency, (3) scalability and (4) the advantages of our hybrid solver. We explore the task of kernel regression for binary supervised classification [33], which requires approximating the solution of $(\lambda I + K)^{-1}x$ during the training step. We use the Gaussian kernel with bandwidth $h$. The model weight $w$ is chosen by solving $w = (\lambda I + K)^{-1}u$, where $u$ is given (the labels). Once $w$ is computed, the label given by $\frac{w}{\|w\|}$ is $\frac{1}{2}$ of $\lambda I$ $\tilde{K}w$. We apply our methods to train this model on real-world datasets employing up to 3,072 x 864 cores and 4,352 KNL cores. The percentage of correct predictions (Acc) is reported in Table I along with the optimal $h$ and $\lambda$ that were found using holdout cross validation.

**Implementation and hardware.** Our experiments were conducted on Lonestar5 (two 12-core, 2.6GHz, Xeon E5-2690 v3 “Haswell” per node) and Stampede (68-core, 1.4GHz, Xeon Phi 7250 “KNL” per node) clusters at the Texas Advanced Computing Center. The theoretical peak performance is 998 GFLOPS per Haswell node and 3,046 GFLOPS per KNL node. Inv-Askit and GSKS are compiled with intel-16 -03 -mavx on Lonestar5 and intel-17 -03 -xMIC-AVX512 on Stampede. All iterative solvers employ a Krylov subspace method (GMRES) from the PETSc library [3]. Specifically, we use modified Gram-Schmidt for re-orthogonalization and employ GMRES CGS refinement. If not specified, KNL experiments use Cache-Quadrant configuration with OMP_PROC_BIND=spread. “T” refers to the total runtime in seconds, and “GFs” refers to the GFLOPS per node.

**Datasets.** We use real-world datasets: COVTYPE (forest cartographic variables); SUSY and HIGGS (high-energy physics) [19]; MNIST (handwritten digit recognition) [5], and MRI (brain MRI) [25]. We also use a 64D synthetic dataset, which is drawn from a 6D Normal distribution and embedded in 64D with additional noise. This set is a dataset with a high ambient but relatively small intrinsic dimension.

**Accuracy metrics and parameter selection.** For the linear solve, we report the relative residual

$$
\epsilon_r = \frac{\|u - (\lambda I + \tilde{K})w\|_2}{\|u\|_2}.
$$

The parameters $h$ and $\lambda$ used in the Gaussian kernel were selected using cross-validation. In Table II we report the parameters we used. Other combinations in Table II are candidates for the cross-validation. Level restriction $L$ is chosen such that the relative error is controlled. In Table II we use $L = 3$, and for experiments in Figure 5 we use $L = 5$ or 7.

### V. Empirical Results

The experiments are labeled #1 to #39 in the tables and figures. We select representative parameter combinations and compare the runtime between [30] and our Algorithm II.4 in Table III. We present single node performance on Haswell and KNL in Table IV. In Figure 4 we present strong scalability and verify the $O(N \log N)$ complexity of our methods (Algorithm II.4). In Table V we compare our

3 We estimate the peak according to the clockrate and the theoretical FLOPS throughput. For 24 Haswell cores, 998 = 2 x 12 x 2.6 x 16. For 68 KNL cores, 3046 = 68 x 1.4 x 32. As a reference, MKL GEMM can achieve 87% on the Haswell node and 69% on the KNL node. We assume two VPPs can dual issue FMA3s [29]. However, Intel processors may have a different frequency while fully issuing FMA3, and the clockrate may drop to 1.0 GHz. This may be the reason why MKL GEMM can only achieve 2.1 TFLOPS on KNL.

| Dataset  | $N$ | $d$ | $h$ | $\lambda$ | Acc   |
|----------|-----|-----|-----|-----------|-------|
| COVTYPE  | 0.1-0.5 | 54  | 0.7 | 3       | 996%  |
| SUSY     | 4.5M | 8   | 0.7 | 10    | 78%   |
| MNIST2M  | 1.6M | 784 | 0.3 | 0      | 100%  |
| HIGGS    | 10.5M | 28  | 90  | 0.01  | 73%   |
| MRI      | 3.2M | 128 | 3.5 | 10    | -     |
| MNIST8M  | 8.1M | 784 | 1.0 | 1.0   | -     |
| NORMAL   | 1-32M | 64  | 19  | 1.0   | -     |

Table II Datasets used in the experiments. Here $N$ denotes the size of the training set, and $d$ is the dimensionality of points in the dataset. The testing sets are disjoint from the training sets. We sample 10K testing points and report the binary classification accuracy in the “Acc” column. $h$ is the bandwidth of the Gaussian kernel used in our experiments. The regression results are produced using the parameters above. Some combinations we used during the cross-validation are presented in details in [7]. MRI, MNIST8M and NORMAL are not used in regression tasks. For the MNIST2M we perform one-vs-all binary classification for the digit ‘3’. All coordinates are normalized to have zero mean and unit variance.
Table III Factorization time comparison in second. Experiments are done on Lonestar5 using 128 nodes (3,072 cores) with adaptive ranks \( s \) selected by \( \tau \) and \( s_{\text{max}} \). COTVTYPE used \( m = 2,048, \kappa = 2,048, s_{\text{max}} = 2,048 \). SUSY used the same combination. MNIST used \( \kappa = 256 \). HIGGS used \( m = 512, \kappa = 1,024 \). NORMAL used \( m = 512, \kappa = 128 \) and \( s_{\text{max}} = 256 \).

Comparison with [36] (Table III). We compare the factorization time between the \( O(N \log^2 N) \) algorithm [36] and our \( O(N \log N) \) algorithms using the same parameters. We only compare the case without level restriction, because [36] does not support this feature. The runtime is directly associated with the rank \( s \). For example, the \( U, V \) matrices in [4] are much larger than those in [36]. Thus, the runtime is also much longer. The overall speedup is about 2–4× due to the \( \log N \) term. Both methods construct exactly the same factorization (up to roundoff errors). Although the speedup is not exactly \( \log N \) due to the prefactors depending on \( s \) and \( d \), we can expect the asymptotic speedup to be \( O(\log N) \). For example, COTVTYPE can only achieve 1.9×, but HIGGS can achieve 3.8× because the problem size is 20× larger. Both methods have \( N \log N \) complexity for the “Solve” operation. For the experiments in Table III, the longest “Solve” operation \( \#9 \) is less than 2 seconds.

Single node performance (Table IV). We conduct a set of single node experiments to show the FLOP rates we achieve and to test some of the models on KNL. On a Haswell node, \#11 reaches 62% (623/998) of the theoretical peak. For a KNL node, \#13 (Cache-Quadrant) is the fastest and achieves 45% (1356/3046). Using MPI on KNL (\( p > 1 \)) is typically slower due to the extra memory operations. Our implementation does not perform very well on the flat memory mode \( \#12 \) and \#10. The memory requirements usually exceed 16GB and as a result \( U \) and \( V \) cannot fit into MCDRAM. We tried manually swapping memory between MCDRAM and DDR4 but this was not as efficient as using the cache memory mode.

Reducing storage (Table IV). In Table IV we report three different schemes for kernel summation II-D: GEMV, GEMM and GSKS. The first scheme takes \( O(sN \log N) \) time and space. The last two schemes evaluate \( K_{\beta \alpha} u \) in hybrid (Algorithm [L6]) with the direct method (Algorithm [LS]). In Figure 5 we report the convergence behavior of iterative solver (Algorithm II.6) and our hybrid factorization. Here the parameter \( \tau \) indicates the relative tolerance of approximation of the kernel matrix \( K \), \( s_{\max} \) is the maximum skeleton size, \( \kappa \) is the number of nearest neighbors used for skeletonization sampling in ASKIT, \( m \) is the leaf node size, and \( L \) is the level restriction.

Comparison with [36] (Table III). We compare the factorization time between the \( O(N \log^2 N) \) algorithm [36] and our \( O(N \log N) \) algorithms using the same parameters. We only compare the case without level restriction, because [36] does not support this feature. The runtime is directly associated with the rank \( s \). For example, the \( U, V \) matrices in [4] are much larger than those in [36]. Thus, the runtime is also much longer. The overall speedup is about 2–4× due to the \( \log N \) term. Both methods construct exactly the same factorization (up to roundoff errors). Although the speedup is not exactly \( \log N \) due to the prefactors depending on \( s \) and \( d \), we can expect the asymptotic speedup to be \( O(\log N) \). For example, COTVTYPE can only achieve 1.9×, but HIGGS can achieve 3.8× because the problem size is 20× larger. Both methods have \( N \log N \) complexity for the “Solve” operation. For the experiments in Table III, the longest “Solve” operation \( \#9 \) is less than 2 seconds.

Single node performance (Table IV). We conduct a set of single node experiments to show the FLOP rates we achieve and to test some of the models on KNL. On a Haswell node, \#11 reaches 62% (623/998) of the theoretical peak. For a KNL node, \#13 (Cache-Quadrant) is the fastest and achieves 45% (1356/3046). Using MPI on KNL (\( p > 1 \)) is typically slower due to the extra memory operations. Our implementation does not perform very well on the flat memory mode \( \#12 \) and \#10. The memory requirements usually exceed 16GB and as a result \( U \) and \( V \) cannot fit into MCDRAM. We tried manually swapping memory between MCDRAM and DDR4 but this was not as efficient as using the cache memory mode.

Reducing storage (Table IV). In Table IV we report three different schemes for kernel summation II-D: GEMV, GEMM and GSKS. The first scheme takes \( O(sN \log N) \) time and space. The last two schemes evaluate \( K_{\beta \alpha} u \) in hybrid (Algorithm [L6]) with the direct method (Algorithm [LS]). In Figure 5 we report the convergence behavior of iterative solver (Algorithm II.6) and our hybrid factorization. Here the parameter \( \tau \) indicates the relative tolerance of approximation of the kernel matrix \( K \), \( s_{\max} \) is the maximum skeleton size, \( \kappa \) is the number of nearest neighbors used for skeletonization sampling in ASKIT, \( m \) is the leaf node size, and \( L \) is the level restriction.

Comparison with [36] (Table III). We compare the factorization time between the \( O(N \log^2 N) \) algorithm [36] and our \( O(N \log N) \) algorithms using the same parameters. We only compare the case without level restriction, because [36] does not support this feature. The runtime is directly associated with the rank \( s \). For example, the \( U, V \) matrices in [4] are much larger than those in [36]. Thus, the runtime is also much longer. The overall speedup is about 2–4× due to the \( \log N \) term. Both methods construct exactly the same factorization (up to roundoff errors). Although the speedup is not exactly \( \log N \) due to the prefactors depending on \( s \) and \( d \), we can expect the asymptotic speedup to be \( O(\log N) \). For example, COTVTYPE can only achieve 1.9×, but HIGGS can achieve 3.8× because the problem size is 20× larger. Both methods have \( N \log N \) complexity for the “Solve” operation. For the experiments in Table III, the longest “Solve” operation \( \#9 \) is less than 2 seconds.

Single node performance (Table IV). We conduct a set of single node experiments to show the FLOP rates we achieve and to test some of the models on KNL. On a Haswell node, \#11 reaches 62% (623/998) of the theoretical peak. For a KNL node, \#13 (Cache-Quadrant) is the fastest and achieves 45% (1356/3046). Using MPI on KNL (\( p > 1 \)) is typically slower due to the extra memory operations. Our implementation does not perform very well on the flat memory mode \( \#12 \) and \#10. The memory requirements usually exceed 16GB and as a result \( U \) and \( V \) cannot fit into MCDRAM. We tried manually swapping memory between MCDRAM and DDR4 but this was not as efficient as using the cache memory mode.

Reducing storage (Table IV). In Table IV we report three different schemes for kernel summation II-D: GEMV, GEMM and GSKS. The first scheme takes \( O(sN \log N) \) time and space. The last two schemes evaluate \( K_{\beta \alpha} u \) in hybrid (Algorithm [L6]) with the direct method (Algorithm [LS]). In Figure 5 we report the convergence behavior of iterative solver (Algorithm II.6) and our hybrid factorization. Here the parameter \( \tau \) indicates the relative tolerance of approximation of the kernel matrix \( K \), \( s_{\max} \) is the maximum skeleton size, \( \kappa \) is the number of nearest neighbors used for skeletonization sampling in ASKIT, \( m \) is the leaf node size, and \( L \) is the level restriction.

Comparison with [36] (Table III). We compare the factorization time between the \( O(N \log^2 N) \) algorithm [36] and our \( O(N \log N) \) algorithms using the same parameters. We only compare the case without level restriction, because [36] does not support this feature. The runtime is directly associated with the rank \( s \). For example, the \( U, V \) matrices in [4] are much larger than those in [36]. Thus, the runtime is also much longer. The overall speedup is about 2–4× due to the \( \log N \) term. Both methods construct exactly the same factorization (up to roundoff errors). Although the speedup is not exactly \( \log N \) due to the prefactors depending on \( s \) and \( d \), we can expect the asymptotic speedup to be \( O(\log N) \). For example, COTVTYPE can only achieve 1.9×, but HIGGS can achieve 3.8× because the problem size is 20× larger. Both methods have \( N \log N \) complexity for the “Solve” operation. For the experiments in Table III, the longest “Solve” operation \( \#9 \) is less than 2 seconds.

Single node performance (Table IV). We conduct a set of single node experiments to show the FLOP rates we achieve and to test some of the models on KNL. On a Haswell node, \#11 reaches 62% (623/998) of the theoretical peak. For a KNL node, \#13 (Cache-Quadrant) is the fastest and achieves 45% (1356/3046). Using MPI on KNL (\( p > 1 \)) is typically slower due to the extra memory operations. Our implementation does not perform very well on the flat memory mode \( \#12 \) and \#10. The memory requirements usually exceed 16GB and as a result \( U \) and \( V \) cannot fit into MCDRAM. We tried manually swapping memory between MCDRAM and DDR4 but this was not as efficient as using the cache memory mode.

Reducing storage (Table IV). In Table IV we report three different schemes for kernel summation II-D: GEMV, GEMM and GSKS. The first scheme takes \( O(sN \log N) \) time and space. The last two schemes evaluate \( K_{\beta \alpha} u \) in
full factorization requires $2^L s N + 2^{2L} s^2$ memory for level restriction $L$, with adaptive $s$ selection. #20, #23 and #26 use the direct factorization on Haswell, and #19, #22 and #25 on KNL. The remaining three runs are done on Haswell using the hybrid method. On Haswell, we observe that the factorization time $T_f$ is about two times longer than #21, #24 and #27. In the factorization phase, both Haswell and KNL do not perform as well as in Table [V] because using adaptive ranks $s$ results in load imbalance. If we further increase $L$, as we need to do in Figure 5, the cost of the full factorization can be 1000× in runtime and 30× in storage.

Applying the hybrid solver to a vector is slower than applying the direct solver, due to the need of iteration. E.g., $T_s$ in #21 is about 20× slower. Yet the overall runtime ($T_f + T_s$) of the hybrid method is still smaller than the direct one. When $L$ is larger, the advantage of the hybrid solver will be higher. For example, in Figure 5 we report results that require $L = 7$. Algorithm II.2 cannot be used: the memory just for $Z$ with $s = 2048$ exceeds 500GB.

**Convergence behavior for solving $\lambda I + \hat{K}$ (Figure 5).**

We report the convergence rate using four different bandwidths with two different methods: (a) unpreconditioned GMRES using ASKIT’s MatVec for $\lambda I + \hat{K}$ (blue line) and (b) our hybrid method Algorithm II.5 (orange line). Each row corresponds to a dataset with a specific $h$. These experiments resemble a cross-validation study in which we vary $\lambda$ in order to improve learning. Across columns, we vary $\lambda$ as $[10^{-2}, 10^{-3}, 10^{-5}]\sigma_1(K)$, where $\sigma_1(K)$ is an estimate, so that the condition number $\kappa$ of $\lambda I + \hat{K}$ is $10^2$, $10^3$ and $10^5$ respectively. We report the relative (to a zero initial guess) Krylov residual $\epsilon_r$ (y-axis) over time (x-axis). The steeper the curve is, the faster the method converges.

The x-axis offset represents setup costs. E.g., in #28, the offset of the blue line ($\approx 140$ sec) is the cost of building the tree and the skeletons (spent in ASKIT). The fixed cost of (b) includes the fixed cost of (a) plus the factorization time.

We see that most of the blue lines are flat when the condition number is around 1E+5, but orange lines still decrease steadily except for #30 (see Table IV for the stability issue). We can observe 10–100× speedup on the “Solve” operations. Overall the hybrid scheme is faster and has more predictable behavior. #30 is detected numerically ill-conditioning of $D$ in our solver. Also notice that in #30 both methods fail to converge.

**VI. CONCLUSIONS**

We have introduced new algorithms for approximately factorizing kernel matrices. We evaluated our algorithms on both real-world and synthetic datasets with different parameters. We conducted analysis and experiments to study the complexity and the scalability of our methods. These experiments include scaling up to 3,072 Haswell cores and 4,352 KNL and exhibit significant speedups over existing methods. The factorization can be very fast. For example, it only takes 10 seconds to factorize a kernel matrix with 32M points in 64D. Our future work will focus on further optimization of our implementation. In particular, we would like to introduce task parallelism in the tree traversal to address the load balancing issue. While adaptive ranks or adaptive level restriction is used, each treenode may have different workload. In this case, scheduling is important to avoid the critical path. Additionally, we plan to address the stability issues mentioned in Table III and explore other possible variants (e.g. sparse off-diagonal blocks).
References

[1] S. Ambikasaran and E. Darve, An $O(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices, Journal of Scientific Computing, 57 (2013), pp. 477–501.

[2] S. Ambikasaran, D. Foreman-Mackey, L. Greenard, D. W. Hogg, and M. O’Neil, Fast direct methods for Gaussian processes and the analysis of NASA Kepler mission data, arXiv preprint arXiv:1403.6015, (2014).

[3] S. Balay et al., PETSc Web page, 2014.

[4] M. Bebendorf, Hierarchical matrices, Springer, 2008.

[5] C.-C. Chang and C.-J. Lin, Libsvm: A library for support vector machines, ACM Transactions on Intelligent Systems and Technology (TIST), 2 (2011), p. 27.

[6] B. Dai, B. Xie, N. He, Y. Liang, A. Raj, M.-F. F. Balcan, and L. Song, Scalable kernel methods via doubly stochastic gradients, in Advances in Neural Information Processing Systems, 2014, pp. 3041–3049.

[7] A. Gittens and M. Mahoney, Revisiting the Nyström method for improved large-scale machine learning, in Proceedings of ICML13, 2013, pp. 567–575.

[8] A. Gray and A. Moore, N-body problems in statistical review, 31 (1989), pp. 221–239.

[9] W. Hackbusch, B. N. Khoromskaia, and E. E. Tyrtyshnikov, Approximate iterations for structured matrices, Numerische Mathematik, 109 (2008), pp. 365–383.

[10] W. W. Hager, Updating the inverse of a matrix, SIAM review, 31 (1989), pp. 221–239.

[11] N. Halko, P.-G. Martinsson, and J. Tropp, Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, SIAM Review, 53 (2011), pp. 217–288.

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

[13] C.-J. Hsieh, S. Si, and I. S. Dhillon, Fast prediction for large-scale kernel machines, in Advances in Neural Information Processing Systems 27, 2014, pp. 3689–3697.

[14] M. Izadi Khaleghabadi et al., Parallel H-matrix arithmetic on distributed-memory systems, (2012).

[15] R. Kondor, N. Teneva, and V. Garg, Multiresolution matrix factorization, in Proceedings of ICML14, 2014, pp. 1620–1628.

[16] R. Kriemann, Parallele Algorithmen fr H-Matrizen, dissertation, Universitt Kiel, 2004.

[17] ______, H-LU factorization on many-core systems, Preprint, Max Planck Institute for Mathematics in the Sciences, 2014.

[18] D. Lee, A. Gray, and A. Moore, Dual-tree fast gauss transforms, Advances in Neural Information Processing Systems, 18 (2006), pp. 747–754.

[19] M. Lichman, UCI machine learning repository, 2013.

[20] Z. Lu et al., How to scale up kernel methods to be as good as deep neural nets, arXiv preprint arXiv:1411.4000, (2014).

[21] 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. 1089–1110.

[22] W. B. March, B. Xiao, S. Tharakkan, C. D. Yu, and G. Biros, A kernel-independent FMM in general dimensions, in Proceedings of SC15, Austin, Texas, Nov 2015.

[23] ______, Robust treecode approximation for kernel machines, in Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Sydney, Australia, August 2015, pp. 1–10.

[24] W. B. March, B. Xiao, C. D. Yu, and G. Biros, An algebraic parallel treecode in arbitrary dimensions, in Proceedings of IPDPS15, Hyderabad, India, May 2015.

[25] Menze et al., The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS), IEEE Transactions on Medical Imaging, (2014), p. 33.

[26] S. M. Omohundro, Five balltree construction algorithms, International Computer Science Institute Berkeley, 1989.

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

[28] S. Si, C.-J. Hsieh, and I. Dhillon, Memory efficient kernel approximation, in Proceedings of The 31st International Conference on Machine Learning, 2014, pp. 701–709.

[29] A. Sodani et al., Knights landing: Second-generation intel xeon phi product, IEEE Micro, 36 (2016), pp. 34–46.

[30] F. G. van Zee and R. A. van de Geijn, Blis: A framework for rapidly instantiating blas functionality, ACM Transactions on Mathematical Software (TOMS), 41 (2015), p. 14.

[31] R. Wang, Y. Li, M. W. Mahoney, and E. Darve, Structured block basis factorization for scalable kernel matrix evaluation, arXiv preprint arXiv:1505.00398, (2015).

[32] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. de Hoop, Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures, SIAM Journal on Scientific Computing, 35 (2013), pp. C519–C544.

[33] L. Wasserman, All of Statistics: A Concise Course in Statistical Inference, Springer, 2004.

[34] 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, no. EPFL-CONF-161322, 2001, pp. 682–688.

[35] C. D. Yu, J. Huang, W. Austin, B. Xiao, and G. Biros, Performance optimization for the k-nearest neighbors kernel on x86 architectures, in Proceedings of SC15, ACM, 2015, pp. 7:1–7:12.

[36] C. D. Yu, W. B. March, B. Xiao, and G. Biros, InvASKIT: a parallel fast direct solver for kernel matrices, in Proceedings of the IPDPS16, Chicago, USA, May 2016.