A Model-Based Algorithm with an Efficient Storage Format for Parallel HSS-Structured Matrix Approximations

Bangtian Liu
CS Department
University of Toronto
Toronto, Canada
bangtian@cs.toronto.edu

Kazem Cheshmi
CS Department
University of Toronto
Toronto, Canada
kazem@cs.toronto.edu

Saeed Soori
ECE Department
Rutgers University
New Jersey, USA
saeed.soori@rutgers.edu

Maryam Mehri Dehnavi
CS Department
University of Toronto
Toronto, Canada
mmehride@cs.toronto.edu

***Abstract*—We present MatRox, a novel model-based algorithm and implementation of Hierarchically Semi-Separable (HSS) matrix computations on parallel architectures. MatRox uses a novel storage format to improve data locality and scalability of HSS matrix-matrix multiplications on shared memory multicore processors. We build a performance model for HSS matrix-matrix multiplications. Based on the performance model, a *mixed-rank* heuristic is introduced to find an optimal HSS-tree depth for a faster HSS matrix evaluation. Uniform sampling is used to improve the performance of HSS compression. MatRox outperforms state-of-the-art HSS matrix multiplication codes, GOFMM and STRUMPACK, with average speedups of $2.8\times$ and $6.1\times$ respectively on target multicore processors.**

I. Introduction

A large class of applications in machine learning, scientific computing, and data analytics involve computations on dense symmetric positive definite (SPD) matrices. Many of these matrices such as the ones in kernel methods in statistical learning [1], Gaussian processes [2], matrix factorizations [3], and integral equations [4] are *structured* (or *low-rank*, or *data-sparse*). Identifying this structure and compressing the low-rank blocks can significantly reduce the storage and computation costs of dense matrix computations. Different low-rank representations have been studied to exploit the structure in dense matrices. Amongst them, Hierarchically Semi-Separable (HSS) algorithms are one of the most widely used.

This paper presents MatRox, a novel algorithm and implementation that improves the performance of HSS matrix computations on multicore architectures. The HSS algorithm first constructs an HSS matrix $\widetilde{K}$ from a dense matrix $K \subset \mathbb{R}^{\mathbb{N}\times\mathbb{N}}$ using low-rank approximation during a *compression* phase. One of the components in compression is a binary tree which we call the *HSS-tree*. As shown in Equation 1, $\widetilde{K}$ is composed of a block diagonal matrix D that stores diagonal blocks of K and low-rank matrices U and V that approximate off-diagonal blocks. The results of compression are used to reduce the computation complexity of operations involving the HSS matrix in an *evaluation* phase. This work focuses on SPD matrices and our evaluation is for matrix-matrix (and matrix-vector) multiplication shown in Equation 2 where $W \subset \mathbb{R}^{\mathbb{N}\times\mathbb{Q}}$.

$$\widetilde{K} = D + UV \tag{1}$$

$$Y = KW \approx \widetilde{K}W \tag{2}$$

Recent work has investigated efficient implementations of the HSS algorithm, specifically the evaluation phase, on parallel multicore architectures [5]–[7]. The HSS-tree represents the computation pattern and the dependency between matrix subblocks during evaluation. To improve the performance of HSS algorithms on parallel processors, most of existing HSS implementations, e.g., GOFMM [5] and STRUMPACK [6], use the HSS-tree as the input for dynamic task scheduling. Thus the parallelism strategies in these libraries differ based on the type of dynamic scheduling used. For example, STRUMPACK builds a queue of tasks by post-order traversal of the HSS-tree and then uses the OpenMP dynamic task scheduler. MatRox also uses the OpenMP dynamic task scheduler but combines it with a queue of tasks that is built by traversing the HSS-tree level-by-level.

Even-though investigating scheduling algorithms is important for improving the performance of HSS algorithms on parallel architectures, in this work, we demonstrate that an efficient data layout, that takes advantage of the inherent data-sparsity in HSS matrices and the pattern of computation in evaluation, can significantly improve the performance and scalability of HSS evaluation on parallel processors. Also, existing libraries require tuning of parameters such as the HSS-tree depth for performance and portability which requires repeatedly executing HSS compression and evaluation phases. We build a theoretical model in MatRox that predicts an optimal depth for the HSS-tree, eliminating tuning overheads for this parameter.

MatRox Overview. MatRox consists of a performance model to analyze the memory bottleneck in HSS algorithms and determine an optimal tree depth (d_{opt}) that coupled with a novel storage format, called *Compressed Data-sparse Storage (CDS)*, significantly improves the performance of the HSS

Algorithm 1: The MatRox Algorithm

Input : K, W, ϵ, arch
Output : Y

/* K is $N \times N$, W is $N \times Q$ */
1 d_{opt} = ModelEval(N, Q, arch)
2 $\widetilde{K}$<CDS> = Compress-to-CDS(K, d_{opt}, ϵ);
/* HSS matrix-matrix multiplication */
3 Y = Evaluate-with-CDS('Mul',$\widetilde{K}$<CDS> , W);
4 return Y

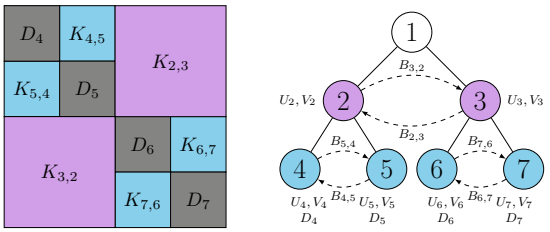


Figure 1: The left figure shows the HSS partitioning of a matrix, and the right figure shows the corresponding HSS-tree.

algorithm on multicore architectures. The model is used with a heuristic to determine an optimal depth d_{opt}. This depth is used in the compression phase in MatRox where uniform sampling [8] with adaptive compression creates $\widetilde{K}$ and stores it in the CDS format. MatRox is shown in Algorithm 1. *ModelEval* in line 1 of the algorithm uses information from the architecture to create a theoretical model of the evaluation phase and to determine the optimum HSS-tree depth. A user-specified accuracy ϵ along with the optimal depth are passed to *Compress-to-CDS* to generate the compressed matrix. The matrix blocks created at compression are stored in the CDS format to follow the computation pattern of HSS evaluation. The compressed matrix in CDS is used in line 3 to efficiently compute the matrix-matrix multiplication in Equation 2.

The paper is organized to discuss preliminaries about the HSS matrix compression and the evaluation process in Section II. Section III-A discusses the ModelEval phase. We will then introduce the CDS storage format in Section III-B and discuss why this format follows the computation pattern of HSS evaluation and how it can improve locality of HSS evaluation and enhance its scalability. The compress-to-CDS phase is discussed in section IV where we demonstrate how compression is implemented in MatRox to generate the compressed matrix in CDS to be used in the evaluation. Our contributions are as follows:

- A novel storage format called CDS that follows the data sparsity and computation pattern of HSS algorithms to improve the performance and scalability of HSS matrix evaluations on shared memory multicore processors.
- A *mixed-rank* heuristic that finds the best performing HSS-tree depth for fast HSS evaluation eliminating tuning overheads. Our heuristic also comes with a theoretical performance model that demonstrates HSS matrix-matrix multiplications are memory bound.
- Implementation of the improved HSS algorithm with the storage format and the mixed-rank heuristic in MatRox. We show that MatRox outperforms GOFMM and STRUMPACK on average $2.8\times$ and $6.1\times$ respectively. The end-to-end HSS algorithm (compression and evaluation) is accelerated with MatRox over GOFMM and STRUMPACK on average $2.3\times$ and $4.1\times$ respectively.

II. Preliminaries

This section briefly explains HSS matrices and how an HSS matrix $\widetilde{K}$ is built from K to be used for approximating KW in Equation 2. We use notation similar to [6]. To construct $\widetilde{K}$, the matrix K is partitioned hierarchically and its off-diagonal blocks are approximated using low-rank matrices as shown in Figure 1. A binary tree, which we call the HSS-tree, is used to represent this hierarchical partitioning. d is used to show the depth of the HSS-tree. The root node depth is 0 and the depth of the HSS-tree shown in Figure 1 is 2.

The index set I_η belongs to the HSS-tree node η and I_T is the set for the root node T where $I_\eta \subset I_T = \{1, \ldots, N\}$. If η is an internal node with children v_1 and v_2 then $I_\eta = I_{v_1} \cup I_{v_2}$. At the leafs of the HSS-tree, the diagonal blocks $D_\eta = K(I_\eta, I_\eta)$ are stored as dense matrices. The off-diagonal blocks $K_{v_1,v2} = K(I_{v_1}, I_{v_2})$, with siblings v_1, and v_2 are approximated using low-rank matrices as

$$K_{v_1,v2} \approx \widehat{U}_{v_1} B_{v_1,v_2} (\widehat{V}_{v_2})^* \tag{3}$$

with tall-skinny matrices $\widehat{U}_{v_1}$ and $\widehat{V}_{v_2}$ that are the bases for the row and column spaces of K_{v_1,v_2}. Complex conjugation is denoted as *. For the internal node η, $\widehat{U}_\eta$ and $\widehat{V}_\eta$ are defined hierarchically as

$$\widehat{U}_\eta = \begin{bmatrix} \widehat{U}_{v_1} & 0 \\ 0 & \widehat{U}_{v_2} \end{bmatrix} U_\eta \text{ and } \widehat{V}_\eta = \begin{bmatrix} \widehat{V}_{v_1} & 0 \\ 0 & \widehat{V}_{v_2} \end{bmatrix} V_\eta \tag{4}$$

while as shown in Figure 1, for the leaf node η, $\widehat{U}_\eta = U_\eta$ and $\widehat{V}_\eta = V_\eta$. Figure 1 shows a two-level HSS partitioning of a matrix along with it's HSS-tree in which D_i, U_i, and V_i are called *generators* for node i and $B_{i,j}$ is the generator for nodes i and j. These generators on HSS will be used to construct matrix $\widetilde{K}$ approximated by HSS. We use a symmetric HSS matrix in this work thus $U_i = V_i$ according to [9].

Compression. The HSS approximation $\widetilde{K}$ of the matrix K is constructed during a compression phase. At compression, a randomized sampling algorithm [10] is often used along with a low-rank matrix factorization such as the interpolative decomposition (ID) [11] to compute generators U, B and V. We will briefly discuss ID and the importance of rank of approximation here; sampling will be elaborated in Section IV. ID approximates a matrix A with rank r by using a subset of r columns of the matrix to interpolate other columns such that $ID(A) = A(:, J)X + E$, where J consists of r column indices, X is the coefficient matrix, $\|E\|$ is $\mathcal{O}(\epsilon)$, and ϵ is the compression error tolerance. The number of columns in J can be adjusted to reduce the compression error. During

Stages	DM	FT	MP	BT
Ops	$Y_i \mathrel{+}= D_i W_i$	L: $\widetilde{W}_j = V_j^* W_j$ I: $\widetilde{W}_j = V_j^* [\widetilde{W}_l; \widetilde{W}_p]$	$\widetilde{Y}_i \mathrel{+}= B_{i,j} \widetilde{W}_j$	L: $Y_i \mathrel{+}= U_i \widetilde{Y}_i$ I: $[\widetilde{Y}_l; \widetilde{Y}_p] \mathrel{+}= U_i \widetilde{Y}_i$

Table I: The evaluation phase stages and operations (Ops). L and I are operations at leaf and inner nodes respectively.

compression, as demonstrated in Equation 3, the off-diagonal blocks, e.g. K_{v_1,v_2}, are approximated. If ID with rank r is used for this approximation, the generators U and V will be tall skinny matrices with r columns and B will of size $r \times r$. In practice, the rank for approximating off-diagonal blocks for an accurate enough HSS approximation is unknown in advance. *Adaptive-rank* methods [12], where the off-diagonal blocks are approximated with different ranks, or a *fixed-rank* method, in which all the off-diagonal blocks are approximated with a tuned fixed rank r, are used to reach the desired accuracy.

Evaluation. The objective of the evaluation phase is to operate on the compressed matrix $\widetilde{K}$, line 3 of Algorithm 1. We organize the evaluation phase into four stages based on [4] as follows: (i) Diagonal Multiplication (*DM*), which performs exact computations on diagonal blocks associated with the HSS leaf nodes, i.e. $Y_i \mathrel{+}= D_i W_i$; (ii) Forward Transformation (*FT*), that operates on the V generators with a tree up-traversal. This stage performs $\widetilde{W}_j = V_j^* W_j$ for leaf nodes and $\widetilde{W}_{j:} = V_j^* [\widetilde{W}_l; \widetilde{W}_p]$ is computed for inner nodes using the results of $\widetilde{W}_l$ and $\widetilde{W}_p$ from the children; (ii) Multiplication Phase (*MP*) which uses the B generators for leaf and inner nodes to compute $\widetilde{Y}_i \mathrel{+}= B_{i,j} \widetilde{W}_j$; (iv) Back Transformation (*BT*), that involves computations on U generators with a tree down-traversal. $[\widetilde{Y}_l; \widetilde{Y}_p] \mathrel{+}= U_i \widetilde{Y}_i$ is performed on an inner node i with children l and p, and results are added to their children; For a leaf node i: $Y_i \mathrel{+}= U_i \widetilde{Y}_i$ is computed and accumulated to the final result. Table I demonstrates all the stages.

Notation. Throughout the paper we use the following notions: N is the number of rows (or columns) of K and the number of rows in W, Q is the number of columns in W, d is the HSS-tree depth, m is the leaf node size, r is the rank for approximation, n_{leaf} is the number of leaf nodes, n_{inner} is the number of inner nodes which are all the nodes except the root and leaf nodes. The relationship between the depth d of an HSS-tree and other variables is: $n_{leaf} = 2^d$, $n_{inner} = 2^d - 2$ and $m = \frac{N}{2^d}$.

III. The Evaluation Performance Model and Evaluation with CDS

The objective of HSS algorithms is to reduce the execution time of operations involving structured dense matrices by compressing K and performing the matrix operation on the compressed matrix $\widetilde{K}$. Improving the performance of the evaluation phase is the main objective of HSS matrix implementations which is also the primary objective of this work. In this section, we propose a novel data layout, called the compressed data-sparse storage format (CDS), to improve the performance and scalability of HSS evaluations for matrix-matrix multiplications on multicore architectures. Prior to discussing CDS, we will derive a performance model for the evaluation phase and introduce MatRox's mixed-rank heuristic for finding an optimal HSS-tree depth for faster evaluation.

Stages	Floating point operations and memory accesses
DM	$C_{DM} = n_{leaf} \times (2m^2Q + mQ)$ $M_{DM} = 8 \times n_{leaf} \times (m^2 + 3mQ)$
FT	$C_{FT} = n_{leaf} \times 2mrQ + n_{inner} \times 4r^2Q$ $M_{FT} = n_{leaf} \times (8m(r+Q) + 8rQ) + n_{inner} \times (16r^2 + 8rQ)$
MP	$C_{MP} = (n_{leaf} + n_{inner}) \times (2r^2Q + rQ)$ $M_{MP} = (n_{leaf} + n_{inner}) \times (8r^2 + 24rQ)$
BT	$C_{BT} = n_{leaf} \times (2mrQ + mQ) + n_{inner} \times (4r^2Q + 2rQ)$ $M_{BT} = n_{leaf} \times (16mQ + 8mr) + n_{inner} \times (16r^2 + 32rQ)$

Table II: Floating point operations and memory accesses in each stage of the evaluation phase.

A. Performance model and optimal depth

The following discusses the performance model for evaluation and the heuristic used in the EvalModel phase of MatRox for determining best depth.

A memory bound evaluation phase. To model the performance and memory access behaviour of the HSS evaluation phase, we use *operational intensity* (OI) from the roof-line model [13] which refers to operations per byte of DRAM access. To compute OI, we divide the number of floating point operations C by memory accesses M after they have been filtered by the cache hierarchy. Data-reuse in the evaluation phase in MatRox only occurs in the FT and BT stages, as shown in table I, and on a small part of the data. We account for this reuse in our model by only once counting accesses to DRAM for these data. Table II and the following shows how we compute C and M:

$$C = C_{DM} + C_{FT} + C_{MP} + C_{BT} \tag{5}$$

$$M = M_{DM} + M_{FT} + M_{MP} + M_{BT} \tag{6}$$

For simplicity, this analysis assumes all the off-diagonal blocks are approximated in rank r. We explain the process of computing C and M for FT as an example. At a leaf node j, the FT stage computes $\widetilde{W}_j = V_j^* W_j$, with $2mrQ$ floating point operations and $8 \times (m(r+Q) + rQ)$ memory accesses. For an inner node j with children l and p, FT performs $\widetilde{W}_{j:} = V_j^* [\widetilde{W}_l; \widetilde{W}_p]$ for inner nodes by using the computed $\widetilde{W}_l$ and $\widetilde{W}_p$ from its children. $4r^2Q$ floating point operations and $8 \times (2r^2 + rQ)$ memory accesses occur for each inner node.

To analyze the behavior of the evaluation phase, we obtain $arch_{minOI} = \frac{arch.peak\_flops}{arch.peak\_BW}$ which is the minimum operation intensity required to achieve maximum performance on the target architecture and compare it to the evaluation algorithm's OI. $arch.peak\_flops$ and $arch.peak\_BW$ are the peak floating point performance and peak memory bandwidth (over all cores) of the architecture respectively. The algorithm is memory-bound when it's OI is less than $arch_{minOI}$ and is compute-bound otherwise. As shown in Figure 2, the

HSS evaluation phase easily becomes memory bound with increasing depth. In the following, we will use the above to determine the best depth for fast evaluation.

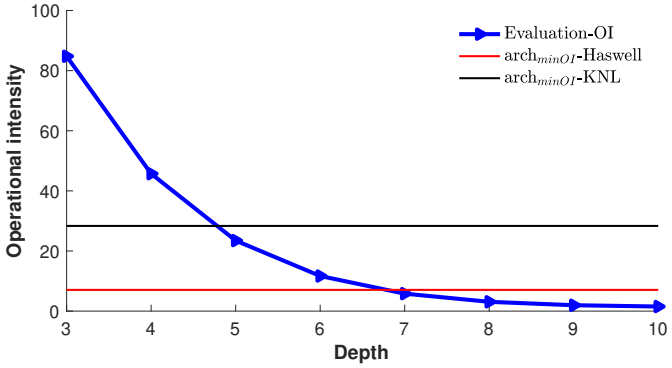


Figure 2: The blue line shows the operational intensity of the evaluation phase for different depths. We use $N = 16K$, $Q = 2048$, and $r = 16$ which are the most common settings in our experiments; other settings also follow the same trend. The horizontal lines show $arch_{minOI}$ for the Haswell and KNL architectures.

Algorithm 2: The Mixed-Rank Heuristic

Input : N, Q, $arch$
Output : d_{opt}

1 **for** $r = 1$; $r \leq r_{max}$; ++r **do**
2 **for** $d = 1$; $d \leq d_{max}$; ++d **do**
  /* Estimate floating point operations */
3 C = estimation_flops(N,Q,d,r)
  /* Estimate memory accesses in bytes */
4 M = estimation_mem(N,Q,d,r)
5 $time_{vec} = \max(\frac{C}{arch.peak\_flops}, \frac{M}{arch.peak\_BW})$
6 **end**
  /* For rank r find the optimal depth */
7 $depth_{vec}$ = $\underset{1 \leq d \leq d_{max}}{argmin}$ time(vec)
8 **end**
  /* Find the most frequent value in vec_d */
9 d_{opt}=freq($depth_{vec}$)
10 return d_{opt}

Finding the optimal depth. In the ModelEval phase of MatRox we execute a mixed-rank heuristic, shown in Algorithm 2, that finds an optimal depth, d_{opt}, for fast evaluation. Line 5 in Algorithm 2 computes a theoretical execution time for evaluation using the following based on the roof-line model:

$$time_{th} = \max(\frac{C}{arch.peak\_flops}, \frac{M}{arch.peak\_BW}) \quad (7)$$

Lines 2-7 find the best performing depth for a specific fixed-rank HSS tree with rank r and store that depth in $depth_{vec}$. The $depth_{vec}$ array is populated with the best performing depth for HSS-trees with different fixed-ranks. Finally in Line 9, the depth that most frequently appears in the $depth_{vec}$ is returned as d_{opt}. In the algorithm d is depth, r is rank, r_{max} is the maximum possible rank of off-diagonal

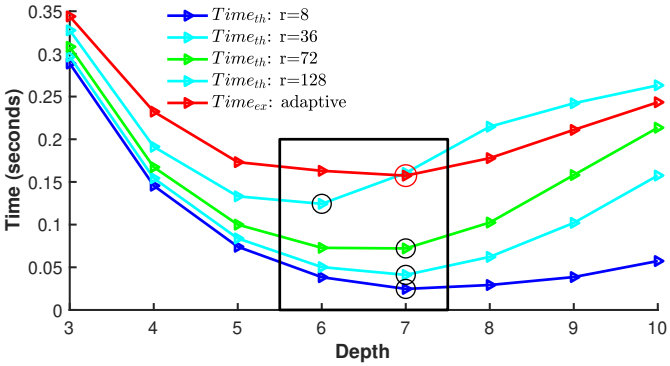


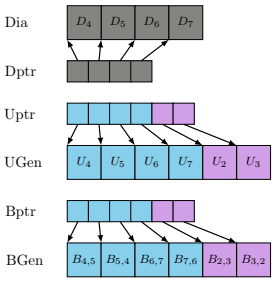
Figure 3: Comparing the mixed-rank heuristic with experimental results. The experiments are executed with 12 cores on Haswell and $N = 16K$, $Q = 2048$. The results are from one kernel matrix generated from *covtype* using the Gaussian kernel function with bandwidth $h = 10$. Other matrices exhibit a similar behavior.

blocks and is bounded by the dimension of these blocks, and d_{max} is the maximum possible depth and is bounded by the matrix dimension. Even-though MatRox is implemented based on adaptive-rank, because the off-diagonal blocks in adaptive-rank methods are approximated with different ranks, our heuristic uses theoretical times from multiple fixed-rank HSS-trees and then chooses the most-frequent best-preforming depth amongst all the fixed-rank HSS-trees. This heuristic works well in practice and finds an optimal rank in all our experiments. Figure 3 demonstrates this by showing the experimental time from MatRox's evaluation phase and the theoretical time for four fixed-rank trees. The circled points are the values stored in $depth_{vec}$ from Algorithm 2 when executed for these ranks. Depth 7 is the most frequent appearing depth in the $depth_{vec}$ array, thus d_{opt} will be 7. As shown in the figure, the experimental results also obtain best performance with depth 7, which shows that the heuristic closely follows the experiments.

We should note that the cost of Algorithm 2, i.e. the EvalModel stage in MatRox, is negligible because the algorithm only operates on scalars and the matrix data is never required for computations in the model.

B. Evaluation with the novel CDS storage format

As discussed in the preliminaries section, the compression of the data-sparse HSS matrix creates sub-matrices to be accessed for evaluation. The existing implementations of HSS evaluation use a linked list to store these sub-matrices. However, a linked list does not guarantee consecutive memory accesses and will negatively effect the performance of the inherently memory-bound HSS evaluation. In the following, we will first present a novel storage format for the evaluation phase with the objective to increase spatial locality. The computation pattern of the DM, FT, MP, and BT stages in evaluation will then be analyzed to demonstrate that evaluation benefits from CDS. MatRox uses this implementation to



(a) The compressed data-sparse (CDS) format

```
1  //DM phase: Exact computation on diagonal blocks
2  #pragma omp parallel for
3  for (j=2^l; j<2^{l+1}; j++)
4      Y[yoffset(j)] += Dia[Dptr[i]] * W[woffset(j)];
5  //FT phase: Bottom to top tree traversal
6  for (i=d; i>1; i--)
7  #pragma omp parallel for
8    for (j=2^i; j<2^{i+1}; j++)
9      if(i == d) // Leaf node
10       FT[foffset(j)] = Trans(UGen[Uptr[j]]) * W[woffset(j)]
11     else
12       FT[foffset(j)] = Trans(UGen[Uptr[j]]) * FT[foffset(left(j))]
13       FT[foffset(j)] = Trans(UGen[Uptr[j]]) * FT[foffset(right(j))]
14 //MP phase: Un each node
15 #pragma omp parallel for
16 for (j=1; j<2^{d+1} - 1; j++) // for every tree node
17   MP[moffset(i)] += BGen[Bptr[j]] * FT[foffset(j)]
18 //BT phase: From top to bottom
19 for (i=1; i<= d; i++)
20 #pragma omp parallel for
21   for (j=2^i; j<2^{i+1}; j++)
22     if(i == d) // Leaf node
23       Y[yoffset(j)] = UGen[Uptr[j]] * MP[moffset(j)]
24     else
25       Y[yoffset(left(j))] = UGen[Uptr[j]] * MP[moffset(j)]
26       Y[yoffset(right(j))] = UGen[Uptr[j]] * MP[moffset(j)]
```

(b) pseudo-code for the evaluation phase

Figure 4: Figure 4(a) shows the CDS storage format for the HSS matrix and the HSS-tree shown in Figure 1. Diagonal blocks D and generators B and U are stored in order in the `Dia`, `BGen`, and `UGen` arrays. Arrays `Dptr`, `Bptr`, and `Uptr` show the starting point of a block in the `Dia`, `BGen`, and `UGen` arrays respectively. Figure 4(b) shows the pseudo-code for evaluation.

improve the data locality of the HSS evaluation on parallel architectures. We will also demonstrate that CDS improves the scalability of evaluation on multicore architectures.

The CDS storage format: We propose the CDS storage format where the HSS matrix, e.g. Figure 1, is stored from a linked list based storage to a layout that improves locality for evaluation. Figure 4(a) shows how the generators and the diagonal blocks of the HSS matrix in Figure 1 are stored in the CDS format. As shown, different diagonal blocks D, and the generators B and U are stored consecutively in arrays `Dia`, `Bgen`, and `Ugen` respectively. `Dptr`, `Bptr`, and `Uptr` point to the starting point of a block in the `Dia`, `Bgen`, and `Ugen` arrays.

The matrix blocks are stored in CDS based on the order of computation to improve spatial locality. By order of computation we are referring to the order which D, B, and U are accessed during parallel execution of the evaluation phase. The HSS-tree represents the dependencies between the sub-matrices of $\widetilde{K}$, thus, the execution pattern is a level-by-level traversal of the HSS-tree and CDS stores the sub-matrices based on this level-by-level traversal. For example, as shown in Figure 4, U_5 and U_6 in Figure 1 are stored in consecutive locations in the `UGen` array. Spatial locality in the evaluation phase is more important than temporal locality since the small sub-matrices accessed during evaluation are seldom reused. For example, while evaluation for an HSS-tree with depth d accesses $2^{d+2}+2^{d+1}+2^{d}-6$ sub-matrices including diagonal blocks, only $2^{d+1}-4$ of these small matrix blocks are reused; data is only reused in the BT and FT stages.

Figure 4(b) shows how we use CDS in the evaluate-with-CDS stage in MatRox. `yoffset`, `woffset`, `moffset`, and `foffset` are used for indexing of *Y*, *W*, `MP`, and `FT` respectively. The evaluation begins with the DM stage where as shown in lines 3–4 of Figure 4(b), diagonal blocks in CDS, the `Dia` array, are accessed. The iterations of this loop are independent of each other and the output of the stage is directly written to the final output `Y` array. The FT phase is shown in lines 6–13 where matrix W is multiplied with the transpose of its corresponding U matrix in the `UGen` array. Since all nodes in a level are independent, all iterations of the inner loop in line 8 run in parallel. Lines 16–17 show the MP stage where the B generators are accessed from `BGen` via the `BPtr` array. The MP stage uses the output of the previous stage that is stored in the `FT` array. The iterations of this loop run in parallel. The last stage is FT where the U matrices are again retrieved from `UGen` in reverse order. Since CDS is a continuous data layout based on the computation order, it works as an ordered ready queue for the scheduler, and thus using a generic scheduler such as `omp parallel` with dynamic scheduling provides a load balanced scheduling.

Scalability: CDS improves the local memory accesses with increasing number of cores to sustain scalability on multicore processors. Large number of main memory accesses can reduce performance on multicore processors [14] and becomes more critical in memory-bound problems such as HSS matrix-matrix multiplication in the evaluation phase. As shown in Figure 5, we use the *average memory access latency* [15] of the evaluation phase for different number of cores (1 to 12) to demonstrate that CDS enhances scalability compared to existing data layouts such as [5]. The average memory access latency, which is a measure for locality, is computed from the L1 cache, TLB, and the last level cache (LLC) misses using *PAPI*. Figure 5 shows the direct relation between average memory access latency improvement in MatRox and the achieved speedup of the evaluation phase compared to GOFMM. The coefficient of determination or R^2 is 0.988 that shows a perfect relation between speedup and average memory access latency. Since CDS follows the HSS-tree level-by-level traversal pattern, the memory access time is reduced as the number of cores scale.

Storage and conversion cost: CDS has low storage and conversion costs. *Storage cost* refers to extra memory locations used in CDS and *conversion cost* is the time that is spent for converting a linked list based storage to the CDS format. Memory needed for storing the D, B, and U blocks is the same as existing storage approaches such as [5]. The only

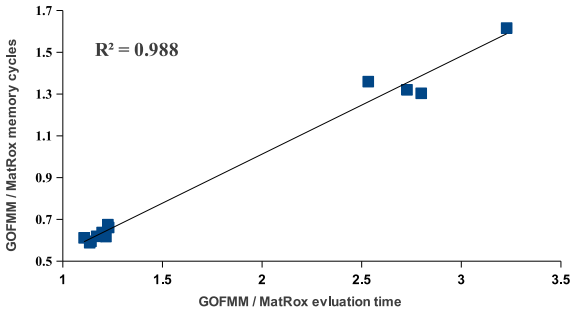


Figure 5: The relation between speedup and average memory latency for the *hepmass* dataset using a Gaussian kernel with bandwidth $h = 10$ on Haswell. Average memory access latency is the average cost of accessing memory in MatRox's code and GOFMM. The coefficient of determination or R^2 of the line is 0.988.

storage cost specific to CDS is the memory required to store the pointer arrays `Dptr`, `Bptr`, and `Uptr` and the storage cost of each is equal to the number of nodes in the HSS-tree, i.e. $O(2N/m)$. CDS has a conversion cost that occurs in the compression stage where the HSS matrix needs to be restored. Since the rank is unknown in adaptive-rank compression, MatRox first stores the data in a temporary buffer, and after the rank of each block is known, it stores the low-rank matrices in the CDS format. The conversion cost for B and U is $O(\frac{2Nr^2}{m})$ and $O(Nr(1+\frac{2r}{m}))$ respectively. The overhead of this conversion is small and is on average 10.8 percent of the compression time for the evaluated matrices in this paper. More details are in Section V.

IV. Compression in MatRox

Compression in HSS algorithms consists of *(i)* permutation, used to reveal the low-rank structure of the matrix, *(ii)* randomized sampling to reduce the computation overhead, and *(iii)* low-rank factorization of sub-sampled matrices. The permutation and low-rank factorization in MatRox's Compress-to-CDS is the same as GOFMM. However, MatRox uses a different sampling method, uniform sampling, which is discussed in this section. In MatRox, the compression phase also has an additional step at the end to store the compressed data into CDS; the overhead for this step was discussed in Section III.B as the conversion cost.

In the following we explain the process of uniform sampling in MatRox. As stated in Section II, compression applies ID to the matrix blocks in each node and computes a factorization of rank-r by expressing each matrix block as a set of its columns [11]. These tall and skinny blocks with rows ranging from $N/2$ to $N-m \approx N$, can lead to a costly ID [6]. Therefore, randomized sampling methods are used a priori to reduce this cost. For a leaf node v_1 and its sibling v_2, consider the matrix block $K_{\bar{v}_1,v_1} = K(I_T - I_{v_1}, I_{v_1})$ where we show the set of all rows except v_1 with $\bar{v}_1$ with index set $I_T - I_{v_1}$. MatRox samples these matrix rows uniformly at random to generate a sampled matrix $K^s_{\bar{v}_1,v_1}$. The number of rows in $K^s_{\bar{v}_1,v_1}$ depends on number of samples. Then, this sub-sampled matrix is approximated by ID as $K^s_{\bar{v}_1,v_1}(:,J_{v_1})(\widehat{U}_{v_1})^*$. By applying the same operation on $K_{\bar{v}_2,v_2}$, we get the approximation $K^s_{\bar{v}_2,v_2}(:,J_{v_2})(\widehat{U}_{v_2})^*$. Finally, we obtain the generator B by $B_{v_1,v_2} = K_{v_1,v_2}(J_{v_1},J_{v_2})$. $\widehat{U}_{v_2} = \widehat{V}_{v_2}$ because of using symmetric matrices in MatRox. The number of samples may vary based on the dataset, but a sampling of size twice the number of columns for each block works well in practice. A similar approach is taken for internal nodes.

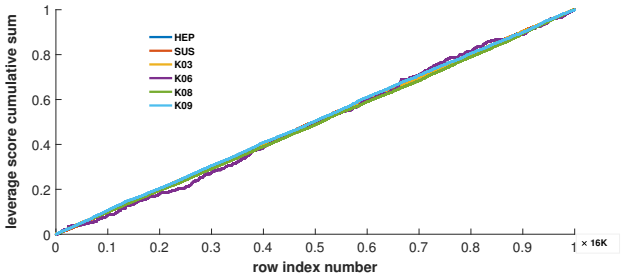


Figure 6: The cumulative sum of normalized leverage scores for the leaf nodes of tested matrices. A uniform distribution has a straight line with slope one.

In Figure 6 we use *leverage scores* to show that uniform sampling performs well for the tested datasets. From [8] row-leverage scores, the Euclidean norm of rows of right singular vectors, can be used as a measure of importance for rows. A uniform distribution of leverage scores demonstrates that uniform sampling will perform well [8]. We compute the leverage scores for rows of random leaf nodes for matrices from both machine learning (*susy*) and scientific applications (*hepmass*, K03, K06, K08, and K09). The cumulative sum for normalized leverage scores is shown in Figure 6. As shown, the row-leverage scores form a uniform distribution thus uniform sampling will work well in practice. Prior work has also shown that uniform sampling performs well for many real-world datasets [16]–[18]; all our experiments which include 20 different matrices from different domains also demonstrate this. Uniform sampling in MatRox has a $O(N)$ overhead which is less than the sampling overhead in STRUMPACK, which uses random projection methods with $O(Nr^2)$, and GOFMM that samples based on nearest-neighbor information with $O(N \log N + N)$. Uniform sampling does not need full access to all elements of the matrix.

V. Experimental Results

We compare the performance of MatRox with STRUMPACK [6] and GOFMM [5], which are considered as two state-of-the-art parallel implementations for HSS algorithms. For compression, the user-specified error tolerance is 1E-5 and the maximum possible rank with adaptive-rank is 256 for all tools. Number of columns in W is Q=2048 and the relative error in evaluation ϵ_f is in the Frobenius norm similar to GOFMM:

$$\epsilon_f = \|\widetilde{K}W - KW\|_F / \|KW\|_F \quad (8)$$

Our dataset includes 20 matrices listed in Table IV. The K02 to K16 datasets are from [5] and the rest are kernel matrices generated using real-world datasets from the UCI repository [19]. K02, K03, K12-K14 are from scientific applications and resemble inverse matrices and Hessian operators from optimization and uncertainty problems. All other matrices are kernel matrices assembled from a kernel function and data points from a real data-set where $K_{ij} = \mathcal{K}(x_i, x_j)$, in which $\mathcal{K}$ is a kernel function and $x_i, x_j \in \mathbb{R}^d$ are data points from the dataset. The K04-K10 are kernel matrices assembled from mostly scientific datasets. Table IV assigns an ID to each matrix. Matrices with ID 16-20 are generated from *covtype*, *susy*, and *hepmass*; *covtype* and *susy* are machine learning datasets. We use a Gaussian kernel [1] with bandwidth $h = 10$ for these datasets. Matrices with ID's 1-18 have a dimension of $16K \times 16K$, the dimension of matrices 19 and 20 is $8K \times 8K$ and $32K \times 32K$ respectively. To generate matrices 16-20 we had to randomly sample from their application data points to fit in memory. GOFMM supports kernel matrices assembled from data points as well as general SPD matrices, thus we use all the datasets for comparison with GOFMM. The kernel function and data points are required inputs to STRUMPACK, thus we compare results from MatRox with STRUMPACK for datasets 16-20 for which we had access to the data points.

MatRox is implemented in C++ with double precision (DP) and uses OpenMP 4.0 [20] for shared memory parallelism. All the code is compiled with icc/icpc 16.0.3. The STRUMPACK and GOFMM are installed using the recommended default configuration. We report the median of 5 executions for all the experiments. The tested architectures are listed in Table III.

Table III: Experimental platform configurations

Processor	Intel Xeon E5-2680 v3	Intel Xeon Phi 7250
Micro-architecture	Haswell	KNL
Frequency	2.5 GHz	1.4 GHz
Physical cores	12	68
Last-level cache	30 MB	34 MB
Peak DP performance	480 Gflops	3264.4 Gflops
Peak memory bandwidth	68 GB/s	115.2 GB/s

In the following, we will first show *end-to-end* (compression and evaluation) results from MatRox with the optimal depth recommended by the mixed-rank heuristic and compare with STRUMPACK and GOFMM with their recommended tuned depths. To demonstrate the performance benefits provided by CDS for evaluation, we then compare MatRox with STRUMPACK and GOFMM using the same HSS-tree depth.

A. MatRox end-to-end results with optimal depth

Table V provides a comprehensive comparison between MatRox, GOFMM, and STRUMPACK for both performance and accuracy, in which all the tools are using their recommended configurations. MatRox uses the mixed-rank heuristic to use an optimal depth d_{opt} during evaluation. As shown in the table, d_{opt} will vary based on the architecture and the matrix dimensions because of the mixed-rank heuristic. The $arch_{minOI}$ on KNL and Haswell is 28.3 and 7.1 respectively, thus, based on Figure 2, because the minimum OI to reach the maximum performance on KNL is higher compared to Haswell, the evaluation algorithm will obtain its best performance with smaller depths on KNL.

Table IV: Test matrices

Matrices	K02-K16	*covtype*	*susy*	*hepmass*
ID	1-15	16	17	18-20

As shown in Table V, MatRox evaluation is on average 3 and 3.5 times faster than GOFMM on Haswell and KNL respectively. Compared to STRUMPACK, MatRox evaluation obtains speedups of $6.7\times$ and $21.0\times$ on Haswell and KNL respectively. The performance improvement in the evaluation phase comes from the optimal depth provided by the mixed-rank heuristic and the proposed storage format. Compression in MatRox is also faster than the other tools because of using uniform sampling. For example, compression with MatRox on Haswell is on average 2.2 and 3.5 times faster than GOFMM and STRUMPACK respectively. If we consider the end-to-end execution of the HSS algorithm which includes both the compression (C) and evaluation (E), MatRox's speedups are $2.3\times$ and $4.1\times$ on average over GOFMM and STRUMPACK on Haswell, $1.7\times$ and $30.2\times$ on average over GOFMM and STRUMPACK on KNL. The compression times reported in MatRox include the conversion time to convert to CDS.

Table V, also shows that the depth recommended by the mixed-rank heuristic does not negatively affect accuracy. As shown in the table, the accuracy of MatRox is close to GOFMM's accuracy and is more accurate than STRUMPACK, which fails to compress matrices 18 and 20 on both KNL and Haswell. MatRox uses the same permutation method [21] as GOFMM and then compresses the off-diagonal blocks with adaptive-rank to reach the user-specified tolerance. STRUMPACK uses a different permutation method, i.e. recursive two-means [22] based on k-means clustering. Thus, the permutation method is more likely to affect accuracy and the observed difference in depth between tools does not have a noticeable effect on accuracy because of using adaptive-rank algorithms.

B. The storage format and scalability

To demonstrate the efficiency of CDS, the scalability and performance of the evaluation phase in MatRox is compared to GOFMM and STRUMPACK. To show the performance benefits of CDS, an HSS-Tree with the same depth is used for all the three tools. This ensures that the sizes and number of matrix subblocks are the same for all tools for fair comparison. Since CDS is used during evaluation, reported times are for the evaluation stage. We show how CDS outperforms the other two libraries for matrices with different data-sparsity and will also demonstrate that CDS leads to a scalable evaluation phase on different architectures.

Figure 7 and Table VI demonstrate the performance benefits of CDS in MatRox compared to GOFMM and STRUMPACK

ID	Haswell										KNL									
	MatRox				GOFMM			STRUMPACK			MatRox				GOFMM			STRUMPACK		
	d_{opt}	C	E	ϵ_f	C	E	ϵ_f	C	E	ϵ_f	d_{opt}	C	E:	ϵ_f	C	E	ϵ_f	C	E	ϵ_f
1	7	0.22	0.10	2E-3	0.42	0.35	9E-4				5	0.99	0.16	7E-4	1.46	0.66	7E-4			
2	7	0.45	0.08	4E-8	0.23	0.31	5E-8				5	1.15	0.11	4E-8	0.72	0.51	5E-8			
3	7	0.39	0.12	3E-5	1.43	0.36	4E-5				5	1.14	0.16	4E-5	2.35	0.58	4E-5			
4	7	0.41	0.20	2E-4	1.47	0.44	9E-5				5	1.35	0.21	7E-5	2.33	0.69	1E-4			
5	7	0.81	0.44	2E-1	1.89	0.82	2E-1				5	2.13	0.42	2E-1	3.32	0.90	2E-1			
6	7	0.38	0.13	4E-6	1.42	0.36	3E-6				5	1.10	0.16	6E-6	2.33	0.56	5E-6			
7	7	0.50	0.26	4E-5	1.54	0.49	4E-5				5	1.51	0.28	4E-5	2.58	0.79	5E-5			
8	7	0.46	0.26	6E-5	1.53	0.51	5E-5	N/A			5	1.57	0.28	5E-5	2.69	0.78	6E-5	N/A		
9	7	0.31	0.11	8E-16	1.45	0.35	1E-15				5	1.04	0.15	9E-16	2.36	0.56	9E-16			
10	7	0.18	0.07	7E-5	0.29	0.31	3E-5				5	0.64	0.10	2E-5	0.95	0.56	2E-5			
11	7	0.21	0.11	4E-4	0.37	0.35	2E-4				5	0.90	0.15	2E-4	0.99	0.66	1E-4			
12	7	0.33	0.14	3E-4	0.56	0.37	3E-4				5	1.12	0.14	2E-4	1.15	0.63	2E-4			
13	7	0.36	0.14	3E-4	0.43	0.38	2E-4				5	1.11	0.14	2E-4	1.13	0.65	2E-4			
14	7	0.90	0.60	2E+0	1.90	0.97	3E+0				5	1.94	0.38	2E+0	3.60	0.93	3E+0			
15	7	1.00	0.61	1E+0	2.19	0.76	1E+0				5	2.05	0.38	1E+0	3.97	0.88	1E+0			
16	7	0.79	0.15	7E-6	0.60	0.54	1E-6	1.28	0.56	1E-5	5	2.69	0.31	4E-7	1.81	0.98	1E-6	45.2	2.22	1E-5
17	7	0.46	0.09	4E-9	0.55	0.46	9E-9	1.31	0.59	1E-5	5	1.61	0.21	3E-9	1.27	0.71	6E-9	45.3	2.45	1E-5
18	7	1.17	0.31	9E-5	1.76	0.99	1E-4	5.14	2.53	1E+0	5	2.47	0.41	8E-5	3.29	0.86	1E-4	97.2	8.43	2E+0
19	6	0.57	0.18	7E-5	0.79	0.50	7E-5	2.36	1.20	7E-1	4	1.78	0.32	6E-5	2.51	0.73	6E-5	32.8	4.65	7E-1
20	8	3.13	0.56	1E-4	3.65	1.97	2E-4	13.5	4.75	3E+0	6	4.78	0.52	1E-4	6.70	1.65	2E-4	261	16.4	5E+0

Table V: End-to-end comparison between MatRox, GOFMM, and STRUMPACK. C and E refer to compression and evaluation. The numbers in the "C" and "E" columns are time in seconds. In MatRox, d_{opt} is determined by using the mixed-rank heuristic.

on Haswell and KNL. MatRox benefits from the better memory layout provided by CDS and is faster than GOFMM and STRUMPACK, $2.8\times$ and $6.1\times$ on Haswell and $3.4\times$ and $14.8\times$ on KNL respectively. The performance benefits obtained from CDS depend on the architecture. Since KNL has higher $arch_{minOI}$ compared to Haswell, better speedups are achieved on KNL over Haswell.

Even-though CDS improves the performance of the evaluation phase for all datasets shown in Figure 7 and Table VI, the amount of speedup achieved differs based on the available data-sparsity in the matrix. We compute the average rank of off-diagonal blocks after the matrix has been compressed (with adaptive-rank) as a measure for the available data-sparsity in the matrix/dataset. The higher the average rank, the less data-sparse the dataset is. The highest reported speedups belong to matrices that are very data-sparse such as matrices 2 and 10 with average ranks of 1 and 2 respectively. Matrices 5, 14, and 15 that have the largest average rank, in order 211, 253 and 255, show relatively lower speedups. As the compressed matrix becomes sparser, operational intensity decreases and thus, memory accesses dominate the execution time. CDS improves the memory access time in MatRox and makes it more efficient for data-sparse matrices.

Figure 8 shows the scalability benefits of CDS for different matrices and architectures. While MatRox scales well on both architectures, GOFMM and STRUMPACK are not scalable or scale weakly when the number of cores increases specially on KNL. As explained in Section III-B and in Figure 5, CDS improves spatial locality and reduces the memory access time with increasing number of cores while scalability in other libraries is limited by memory access times.

As discussed in Section III-B, the compression phase of MatRox has an additional stage that converts the linked-list based storage to CDS which we call the conversion time. Figure 9 shows the ratio of conversion to the entire compression time on Haswell. As shown, the conversion cost is small and is on average 10.8 percentage of the entire compression phase on Haswell; the same trend is also observed on KNL. Also, often the datasets are compressed once with HSS compression and then used for repeated evaluations thus this cost will amortize.

VI. Related work

Hierarchical matrices are used to approximate matrix computations in almost linear complexity. Hackbusch first introduced $\mathcal{H}$-matrices [23], [24], in which the matrix is partitioned hierarchically using a cluster tree and then parts of the off-diagonal blocks are approximated. Later, $\mathcal{H}^2$-matrices were introduced [25] which use nested basis matrices, to further reduce the computational complexity of dense matrix computations. Many low-rank formats exist and most are classified as $\mathcal{H}$- or $\mathcal{H}^2$-matrices. For example, Hierarchically Off-Diagonal Low-Rank (HODLR) [26] which approximates all off-diagonal blocks in low-rank is considered as a specific subclass of $\mathcal{H}$-matrices. HSS is similar to HODLR but uses nested basis matrices and thus is a subclass of $\mathcal{H}^2$-matrices.

Numerous algorithms are studied for the compression of hierarchical matrices some of which include truncated SVD, rank-revealing QR [27], rank-revealing LU [28], CUR factorization [29], Nystrom method [30], adaptive cross approximation [31], and interpolative decomposition (ID) [11]. MatRox uses ID in its compression phase. Randomized algorithms [10] are frequently used for faster low-rank approximations during compression. For example, Martinsson [11] uses randomized sampling for the construction of HSS matrices, in which elements are sampled from the matrix with matrix-vector multiplications. ASKIT [21] and GOFMM implement neighbor-based importance sampling and STRUMPACK [6]

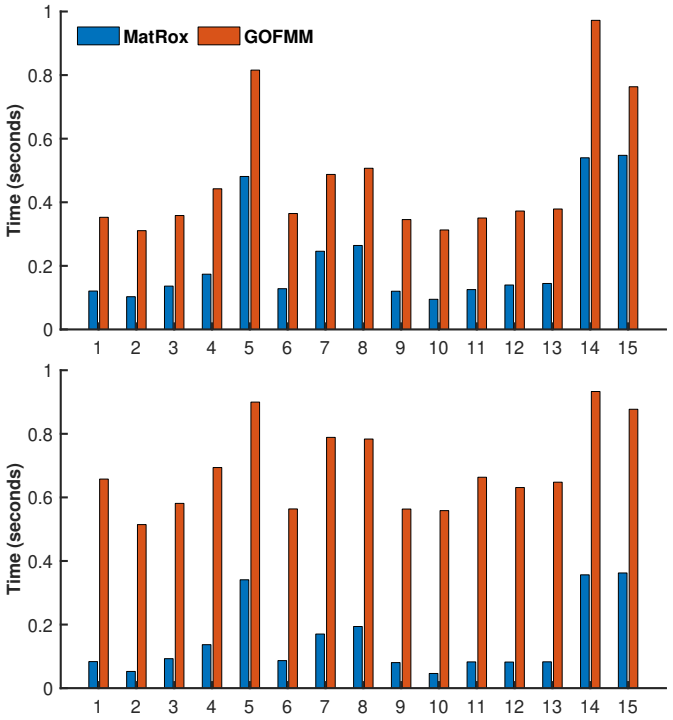


Figure 7: The effect of CDS on the performance of evaluation on Haswell (top) and KNL (bottom). All times are reported for the same HSS-tree depth.

	Haswell			KNL		
ID	MatRox	GOFMM	STRUMPACK	MatRox	GOFMM	STRUMPACK
16	0.16	0.54	0.57	0.25	0.98	1.74
17	0.10	0.46	0.66	0.17	0.71	1.84
18	0.28	0.99	1.90	0.37	0.86	6.85
19	0.16	0.50	0.94	0.30	0.73	3.78
20	0.50	1.97	3.68	0.53	1.65	13.4

Table VI: CDS performance for matrices 16-20 (with times too large to fit in Figure 7). Numbers show the evaluation time in seconds. All implementations use the same HSS-tree depth.

uses random projection [8]. Uniform sampling has been shown to perform well for many real-world problems [16], [17] and is used in MatRox.

During the evaluation, hierarchical matrices can be used to reduce the complexity of numerous matrix operations. The frontal matrices in multifrontal methods often have a low-rank structure [3] thus hierarchical matrices have been studied to accelerate matrix factorization and develop fast solvers in work such as [9], [26], [32]. $\mathcal{H}$-matrices are used with LDL^T to build robust preconditioners for GMRES [33]. Ghysels *et al.* [34] introduces a parallel and fully algebraic preconditioner based on HSS. Other work has improved the complexity of matrix operations such as matrix inversion [35] and matrix-vector/matrix multiplication [32]. The evaluation phase in STRUMPACK and GOFMM supports matrix-vector and matrix-matrix operations which we use for comparison.

Hierarchical matrix algorithms have been implemented on various platforms ranging from shared memory [5]–[7], dis-

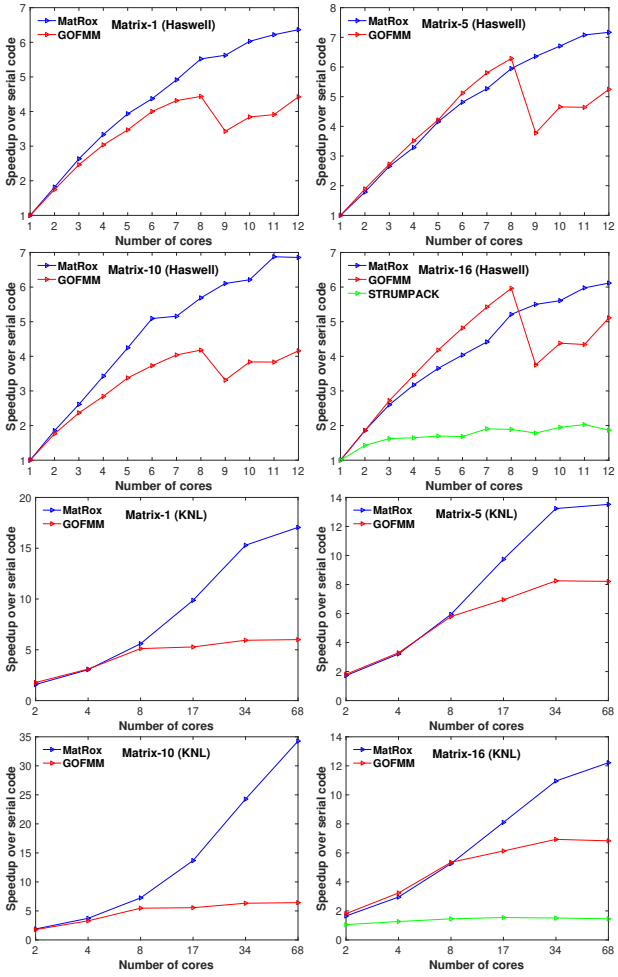


Figure 8: The effect of the CDS format on the scalability of the evaluation phase on Haswell (the top four figures) and KNL (the bottom four figures).

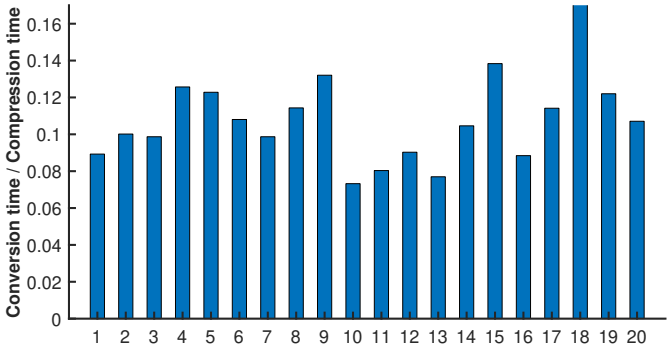


Figure 9: The conversion time to compression time ratio for CDS on Haswell.

tributed memory [12], [36], [37], and many-core architectures such as GPUs [38]. ASKIT and GOFMM provide a parallel fast multipole method on multicore processors. STRUMPACK [6], [12] and Hsolver [39] use HSS for accelerating multifrontal factorization on both distributed architectures and

multicore processors. STRUMPACK also supports matrix-matrix multiplication on distributed architectures [12]. Existing parallel codes take advantage of tree parallelism for an efficient parallel implementation. Particularly, existing multicore frameworks such as GOFMM propose novel scheduling in combination with tree traversals. However, MatRox focuses on improving the memory access time of HSS matrix-matrix multiplication on multicore processors by proposing a performance model and a new storage format.

VII. Conclusion

We introduce MatRox to improve the performance of HSS algorithms on shared memory multicore architectures. We propose a novel storage format to improve the performance and scalability of HSS evaluation. Our mixed-rank heuristic finds the optimal HSS-tree depth for faster HSS evaluation eliminating the need to tune for best depth. Uniform sampling is also used for fast compression. Our experiments show MatRox outperforms state-of-the-art HSS libraries on multicore processors.

Acknowledgement

This work was supported in part by the grants NSF CCF-1814888 and NSF CCF-1657175. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) [40], which is supported by National Science Foundation grant number ACI-1548562.

References

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

[2] S. Börm and J. Garcke, "Approximating gaussian processe with $\mathcal{H}^2$-matrices," in *European Conference on Machine Learning*. Springer, 2007, pp. 42–53.

[3] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam, "On the numerical rank of the off-diagonal blocks of schur complements of discretized elliptic pdes," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 5, pp. 2261–2290, 2010.

[4] W. Hackbusch, *Hierarchical matrices: algorithms and analysis*. Springer, 2015, vol. 49.

[5] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, "Geometry-oblivious fmm for compressing dense spd matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 53.

[6] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, "An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S358–S384, 2016.

[7] R. Kriemann, "Parallel-matrix arithmetics on shared memory systems," *Computing*, vol. 74, no. 3, pp. 273–297, 2005.

[8] M. W. Mahoney *et al.*, "Randomized algorithms for matrices and data," *Foundations and Trends® in Machine Learning*, vol. 3, no. 2, pp. 123–224, 2011.

[9] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "Fast algorithms for hierarchically semiseparable matrices," *Numerical Linear Algebra with Applications*, vol. 17, no. 6, pp. 953–976, 2010.

[10] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

[11] P.-G. Martinsson, "A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1251–1274, 2011.

[12] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, p. 27, 2016.

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[14] X.-H. Sun and Y. Chen, "Reevaluating amadahl's law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183–188, 2010.

[15] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2017.

[16] S. Kumar, M. Mohri, and A. Talwalkar, "Sampling methods for the nyström method," *Journal of Machine Learning Research*, vol. 13, no. Apr, pp. 981–1006, 2012.

[17] M. B. Cohen, Y. T. Lee, C. Musco, C. Musco, R. Peng, and A. Sidford, "Uniform sampling for matrix approximation," in *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*. ACM, 2015, pp. 181–190.

[18] A. Gittens and M. W. Mahoney, "Revisiting the nyström method for improved large-scale machine learning," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 3977–4041, 2016.

[19] A. Asuncion and D. Newman, "Uci machine learning repository," 2007.

[20] A. OpenMP, "Openmp 4.0 specification," 2013.

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

[22] E. Rebrova, G. Chavez, Y. Liu, P. Ghysels, and X. S. Li, "A study of clustering techniques and hierarchical matrix formats for kernel ridge regression," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018.

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

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

[25] W. Hackbusch, B. Khoromskij, and S. Sauter, "On h2-matrices: Lectures on applied mathematics," 2000.

[26] A. Aminfar, S. Ambikasaran, and E. Darve, "A fast block low-rank dense solver with applications to finite-element matrices," *Journal of Computational Physics*, vol. 304, pp. 170–188, 2016.

[27] T. F. Chan, "Rank revealing qr factorizations," *Linear algebra and its applications*, vol. 88, pp. 67–82, 1987.

[28] L. Miranian and M. Gu, "Strong rank revealing lu factorizations," *Linear algebra and its applications*, vol. 367, pp. 1–16, 2003.

[29] M. W. Mahoney and P. Drineas, "Cur matrix decompositions for improved data analysis," *Proceedings of the National Academy of Sciences*, pp. pnas–0 803 205 106, 2009.

[30] C. K. Williams and M. Seeger, "Using the nyström method to speed up kernel machines," in *Advances in neural information processing systems*, 2001, pp. 682–688.

[31] M. Bebendorf and S. Rjasanow, "Adaptive low-rank approximation of collocation matrices," *Computing*, vol. 70, no. 1, pp. 1–24, 2003.

[32] S. Chandrasekaran, M. Gu, and T. Pals, "A fast ulv decomposition solver for hierarchically semiseparable representations," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[33] B. Engquist and L. Ying, "Sweeping preconditioner for the helmholtz equation: hierarchical matrix representation," *Communications on pure and applied mathematics*, vol. 64, no. 5, pp. 697–735, 2011.

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

[35] P.-G. Martinsson and V. Rokhlin, "A fast direct solver for boundary integral equations in two dimensions," *Journal of Computational Physics*, vol. 205, no. 1, pp. 1–23, 2005.

[36] W. B. March, B. Xiao, S. Tharakan, D. Y. Chenhan, and G. Biros, "A kernel-independent fmm in general dimensions," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 2015, pp. 1–12.

[37] W. B. March, B. Xiao, and G. Biros, "Askit: Approximate skeletonization kernel-independent treecode in high dimensions," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. A1089–A1110, 2015.

[38] W. B. March, B. Xiao, D. Y. Chenhan, and G. Biros, "An algebraic parallel treecode in arbitrary dimensions," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 571–580.

[39] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, and M. V. De Hoop, "A parallel geometric multifrontal solver using hierarchically semiseparable structure," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 3, p. 21, 2016.

[40] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "Xsede: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.