jh190043-NAHI

Hierarchical Low-Rank Approximation Methods on Distributed Memory and GPUs

Rio Yokota (Tokyo Institute of Technology)

Abstract

Hierarchical low-rank approximation of dense matrices can reduce the complexity of matrix multiplication and factorization from $\mathcal{O}(N^3)$ to $\mathcal{O}(N \log^2 N)$ while trading accuracy for speed. Modern processors are equipped with low-precision arithmetic, which yield much higher throughput if low accuracy can be tolerated. On such processors, performing exact $O(N^3)$ dense linear algebra operations through the use of LAPACK and BLAS libraries is a waste of Flops. Our goal is to replace these exact dense linear algebra libraries with our hierarchical low-rank approximation library. The functions needed for this are the capability to perform LU and QR factorization on GPUs and distributed memory. For FY 2019 we focused on the following critical steps to achieving this goal; Implement nested basis version of $\mathcal{H}$-matrix ($\mathcal{H}^2$-matrix), Use of runtime for $\mathcal{H}$-LU on GPU, Batched randomized SVD on GPUs, Extending the $\mathcal{H}$-matrix code to QR factorization.

1 Basic Information

1.1 Collaborating JHPCN Centers

The University of Tokyo
Information Technology Center

Tokyo Institute of Technology
Global Scientific Information and Computing Center

Hokkaido University
Information Initiative Center

Kyoto University
Academic Center for Computing and Media Studies

Nagoya University
Information Technology Center

1.2 Research area

Very large-scale numerical computation

1.3 Roles of Project Members

Rio Yokota (Tokyo Institute of Technology) Low-rank approximation using FMM and its GPU-MPI implementation

Ichitaro Yamazaki (Sandia National Laboratories) Development of distributed memory runtime ParSEC, and blocked BLAS library for GPU

Akihiro Ida (University of Tokyo) Feature extension of hybrid MPI/OpenMP $\mathcal{H}$-matrix code HACApK, and its integration with ParSEC and block MAGMA

Takeshi Iwashita (Hokkaido University) Application of HACApK to boundary integral solvers for electromagnetics, and optimization of $\mathcal{H}$-matrix-vector product

Takayuki Aoki (Tokyo Institute of Technology) Application of HACApK to Poisson solvers for multiphase flows

Satoshi Oshima (Nagoya University) GPU implementation of HACApK and integration with MAGMA

Tasuku Hiraishi (Kyoto University) Dynamic load-balancing of HACApK

Kengo Nakajima (University of Tokyo) Extend capability of HACApK within the ppOpen-HPC framework

Jack Dongarra (University of Tennessee) Development of distributed memory runtime ParSEC, and blocked BLAS library for GPU

2 Purpose and significance of Research

The purpose of this research is to develop a scalable and highly optimized open source library for hierarchical low-rank approximation of dense matrices ($\mathcal{H}$-matrices). Such large dense matrices naturally appear in electromagnetic, seismic, quantum, and fluid simulations, as well as Bayesian inference and machine learning applications. Unlike their dense counterparts which require $\mathcal{O}(N^3)$ time and $\mathcal{O}(N^2)$ memory, $\mathcal{H}$-matrices can perform matrix multiplication and factorization in $\mathcal{O}(Nlog^2N)$ time and $\mathcal{O}(N)$ memory, have controllable arithmetic intensity, have asynchronous communication, and can exploit deep memory hierarchy, which makes them an ideal solver/preconditioner for the Exascale era.

In the previous JHPCN project we have ported our $\mathcal{H}$-matrix library HACApK to GPUs and used batched MAGMA to further accelerate the GPU implementation. The original HACApK code was only capable of performing $\mathcal{H}$-matrix vector multiplications, but we have extended it to perform matrix-matrix multiplications, and LU factorization. We have also developed a lattice $\mathcal{H}$-matrix method, which uses BLR structure for the global data structure, while using the $\mathcal{H}$-matrix structure for the local data structure, by embedding $\mathcal{H}$-matrices into the BLR blocks. However, the scalability of the multi-GPU implementation is still far from ideal.

In the proposed JHPCN project we plan to improve the scalability of the multi-GPU $\mathcal{H}$-LU factorization code, and also extend HACApK to handle QR factorization as well. We will also investigate the $\mathcal{H}^2$-matrix format to reduce the communication by exploiting the nested bases. Furthermore, we will introduce runtime systems such as StarPU/OmpSs to handle the complex data dependencies in the hierarchical matrix structure, while maximizing concurrency during the execution on heterogeneous architectures.

3 Significance as JHPCN Joint Research Project

Hardware architecture is now moving towards low-precision arithmetic, backed by the increasing demand from the machine learning field. When such low-accuracy can be tolerated, exact dense linear algebra operations become unnecessary, and libraries such as BLAS and LAPACK, which are at the heart of HPC applications, can be replaced by hierarchical low-rank ($\mathcal{H}$-matrix) libraries that effectively do the same work in near linear time. There is still ample room for investigation regarding the use of such low-precision in scientific computing applications, where methods such as iterative refinement have recently gained interest. $\mathcal{H}$-matrices can be used as a scalable preconditioner for such problems, and we aim to quantify the advantage over existing state-of-the-art methods in this JHPCN project. Furthermore, batched operations on GPUs are becoming popular and libraries such as MAGMA and CUBLAS are providing low-level functions that can process many small dense matrix operations in large batches. $\mathcal{H}$-matrices can benefit greatly from such batched dense linear algebra libraries, and in doing so will be able to extract a large portion of the performance of the latest GPU and many-core architectures including Ten-

sorCores. Since libraries like MAGMA and CUBLAS are optimized to use TensorCores, we do not have to do the implementation ourselves.

4 Outline of Research Achievements up to FY2018

Up to FY2018 we have tackled various problems regarding hierarchical low-rank approximation and its parallel implementation. There are various derivatives of hierarchical low-rank approximation methods such as; BLR, HODLR, HSS, $\mathcal{H}$-matrix, and $\mathcal{H}^2$-matrix. We started from the most basic variant – BLR, which uses low-rank off-diagonal blocks, but not a hierarchical matrix. We started with the most basic operations such as matrix-vector and matrix-matrix multiplication. This was extended during FY2016 to LU factorization and implemented in OpenMP and MPI. In FY2017, we extended the matrix format to more complex HSS and $\mathcal{H}$-matrix structures, and extended the implementation to GPUs for the matrix-vector multiplication. We utilized batched MAGMA operations to process the matrix-vector multiplication efficiently on GPUs. In FY2018, we further extended the implementation of the LU factorization to multiple-GPUs using a hybrid MPI + OpenMP + CUDA code.

5 Details of FY2019 Research Achievements

The four main goals for the fiscal year 2019 are

1. Implement nested basis version of $\mathcal{H}$-matrix ($\mathcal{H}^2$-matrix)
2. Use of runtime for $\mathcal{H}$-LU on GPU
3. Batched randomized SVD on GPUs
4. Extending the $\mathcal{H}$-matrix code to QR factorization

During the first half of FY2019, we were able to complete all goals except for the first one ($\mathcal{H}^2$-matrix). In the second half of FY2020, we were able to complete the remaining nested basis implementation.

5.1 Implement nested basis version of $\mathcal{H}$-matrix

For the extension to nested basis, we first started from an implementation that uses uniform basis, where the bases are shared among the rows and columns, but the matrix is not hierarchical. This allowed us to experiment with different code structures using our C++ implementation [10]. Once the uniform basis is finished, we can add the upper levels of the $\mathcal{H}^2$-matrix. Our code is written so that the LU and QR operations, the nested and non-nested basis, the CPU, GPU, and MPI implementation can be developed separately, and can be integrated to achieve a combination of these features without any extra work.

5.1.1 $\mathcal{H}^2$-matrix construction

For matrices generated using a kernel function, the separation in the clusters creates low-rank blocks. As an analogy to the Fast Multipole Method, the construction of $\mathcal{H}^2$-matrices also uses the distances between clusters to determine the block admissibility, and the admissible blocks formed from the same cluster are sharing the same basis. In our implementation, to find the basis of any designated cluster, we actively generates another arbitrary accompanying cluster, that has a distance far enough from the designated cluster by checking the admissibility condition. By performing an interpolative decomposition on the block formed from the designated cluster and the accompanying cluster, we can obtain the uniform basis that can be shared among all blocks formed from that cluster. Furthermore, when constructing the Uniform basis Block Low-Rank matrices, it is only matters of compressing the admissible blocks from the already found bases. However, the construction of $\mathcal{H}^2$-matrices also includes sampling the basis of combined lower-level clusters and the formation of a hierarchy on cluster bases themselves. In our implementation, we still used the same interpolative decomposition method on higher-level clusters, which could potentially be harmful to the performance due to the increased sample block size.

5.1.2 $\mathcal{H}^2$-matrix addition and multiplication

When performing $\mathcal{H}^2$-arithmetics, two key operations are split and collect. The split operation takes a compressed low-rank matrix block and projects it to lower levels, using only the transfer matrices stored in the cluster basis hierarchy; the collect operation is the opposite of the split operation, which forms a compressed low-rank block from lower-level blocks, not mattering if the lower-level blocks are stored in dense or low-rank formats. When adding or subtracting two $\mathcal{H}^2$-matrices with the assumption that the cluster bases are the same, we actively convert the other matrix into the same hierarchical structure as the accumulator matrix, by using the split and collect operations accordingly. On the other hand, when performing multiplications involving $\mathcal{H}^2$-matrices, we still uses the accumulator as the structural reference, and the same assumption that the two multiplicands shares the inner bases among them while the outer bases with the accumulator. As the split and collect operations have very low computational costs, we actively performs a forward transformation of the two multiplicand matrices, that approximates all blocks from splitting to lower levels and collecting the lower level blocks to approximate higher levels. Whenever the accumulator needs the approximated higher-level blocks of the multiplicands, it does require additional computations. As a contrast to the forward transformation, when the multiplicand is highly compressed but the accumulator is not, the backward transformation uses only split, to project a low-rank intermediate multiplication result down to the very bottom levels. With the assistance from the two transformations, the $\mathcal{H}^2$-matrix multiplication could be finished quite efficiently as well.

5.1.3 $\mathcal{H}^2$-matrix arithmetics with basis updates

The previous section addressed highly efficient ways of performing $\mathcal{H}^2$-arithmetics, but not all $\mathcal{H}^2$-arithmetics can satisfy the assumption of the basis shared among operands, such as the multiplications and additions happened in LU factorization. In such cases, we are also exploring the methods of updating the accumulator basis as the computation goes, while keeping the shared basis structure intact, and to both minimize the computational costs and to improve accuracy. In our implementation, we have left rooms for the basis hierarchy that can take additional fill-ins. The basis update only happens in low-rank accumulations, as it checks if the current bases could compress the other block within the specified accuracy limits, without appending additional vectors. Although the accuracy checks and the basis updates have relatively low computational costs, the large number of required low-rank accumulations happening potentially leads to greater computational time required, when comparing to the previous method which does not include basis update. However, the additional computational effort has proved itself effective in accuracy improvements, during LU factorization of Uniform basis Block Low-rank matrices, that the relative error has improved from 1E-5 (no basis update) to 1E-10 (with basis update).

5.2 Use of runtime for $\mathcal{H}$-LU on GPU

In the previous JHPCN project we ported HACApK to GPU and also extended it to handle $\mathcal{H}$-LU factorization. However, these efforts had not been combined, and we did not have an $\mathcal{H}$-LU factorization on GPUs. For the $\mathcal{H}$-matrix-vector multiplication, we were able to make use of the batched dense linear algebra libraries because there were no dependencies between the tasks. Unfortunately, the batched libraries cannot be used if there are dependencies between the tasks in the batch, which is the case for LU factorization. To solve this problem out plan was to resort to runtime systems like StarPU and OmpSs to handle these dependencies in the $\mathcal{H}$-LU algorithm and efficiently schedule them on the GPU.

After experimenting with StarPU and OmpSs, we reached the conclusion that the overhead of both StarPU and OmpSs were too large to be used at the granularity re-

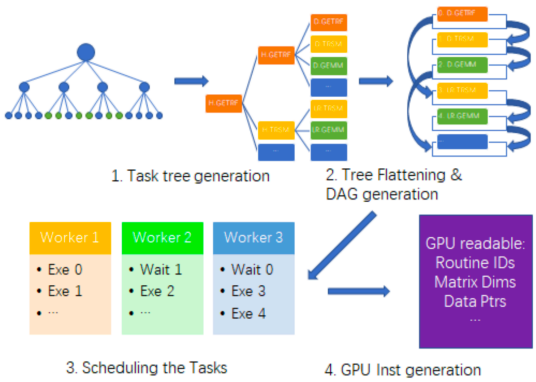

Fig. 1 Illustration of the task scheduling of $\mathcal{H}$-LU on the host

quired for our $\mathcal{H}$-LU operations. We could superficially increase the compute intensity by making the leaf blocks of the dense diagonal blocks larger, or by making the ranks large for the off-diagonal low-rank blocks. However, this will lead to suboptimal baseline performance, and its GPU implementation will not be the ideal solution. Therefore, we developed our own lightweight DAG scheduler, which performs much better than StarPU or OmpSs [7]. Our DAG (Directed Acyclic Graph) scheduler runs on the CPU and queues the tasks to be executed on the GPU. Then we dispatch a single GPU kernel that does the entire $\mathcal{H}$-LU operation.

Our scheduling algorithm consist of the following four steps, as shown in Fig. 1.

1. Task tree generation
   CPU traverses the hierarchical structure and determines the operations needed to perform on each block to complete the LU factorization correctly, without modifications on the actual data.
2. Tree Flattening & DAG generation
   Tree flattening takes the tree of tasks generated, and gets rid of intermediate hierarchical nodes, which are fully represented by their dense and low-rank children. Data dependency checks are then performed between each task, and the result is stored as a DAG.
3. Scheduling the Tasks
   With the tasks and their data dependency information, the scheduler uses 2 different heuristics to create a static scheduling: which task to fetch & which worker to schedule to. Additional trimming and the estimated FLOPS are also used to assist the scheduler to further improve device utilization.
4. GPU task queue generation
   The last task of the CPU is translating the scheduling and the task information to the GPU correctly. Data pointers are batched together into an array, and BLAS-like and low-rank routines are enumerated. Along with the matrix dimensions and pointer offsets, each worker gets an integer array to represent the tasks being assigned to them.

As all tasks are properly batched, there is no need for GPU to know the hierarchical structure, which implies no more recursion inside the kernel. In general, the kernel is made of 2 components; kernel level controlling functions and thread-block level routines. Additionally, warp-level intrinsics are used to optimize the thread-block level routines.

On the kernel level, it is sometimes necessary to deliberately halt one or several SMs in order to correctly meet data dependency requirements until another SM finishes its work. However, as NVIDIA does not provide such functions in the CUDA programming model, we allocated space in the global memory as communication space for the SMs, and adopted a spin-lock mechanism. Instead of halting, SMs each assigns 1 thread that actively checks the state of the lock that it is waiting. In order to save the memory bandwidth, and possess minimal impact to other busy SMs, a waiting time hint is also provided by the CPU, correlating to the estimated FLOPS that needs to be completed.

On the thread-block level, several LAPACK-like routines and Low-rank routines are implemented as device functions in the kernel. A schematic of the GPU execution is shown in Fig. 2 Each device

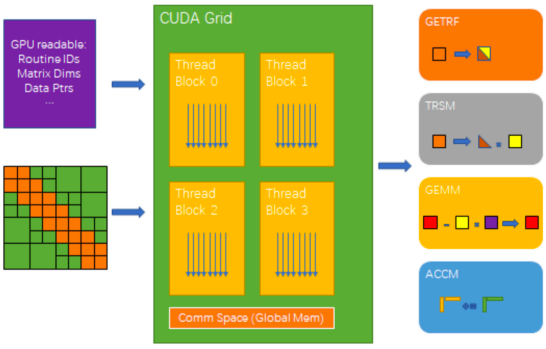

Fig. 2 Illustration of the CUDA grid execution of device function LAPACK kernels

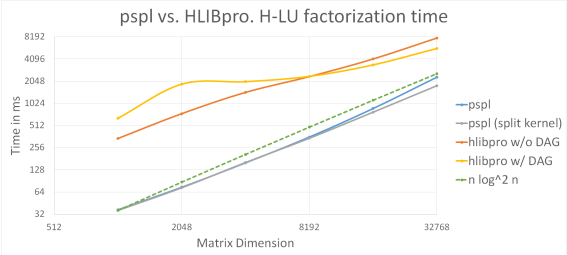

Fig. 3 Comparison between our GPU implementation (pspl) and an existing CPU implementation (HLibPro)

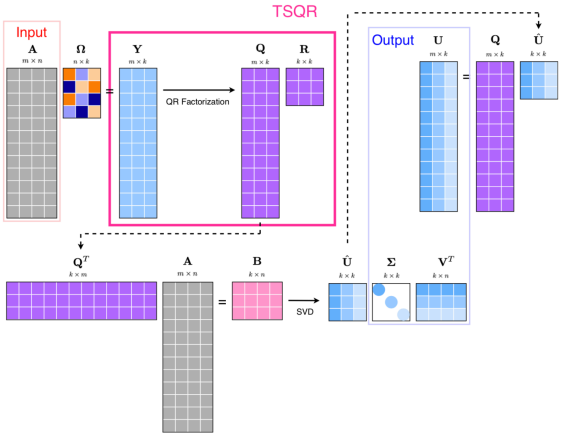

Fig. 4 Schematic of randomized SVD

function takes all threads assigned to 1 SM to execute, and utilizes the full amount of shared memory and L1 cache. Inside each device function, vectorized memory I/O and warp shuffling techniques are used to further optimize each routines.
Our entire code consists of only four basic routines:

- GETRF: LU decomposition
- TRSM: Triangular solve of linear systems
- GEMM: General Matrix-matrix multiplication
- ACCM: Accumulation of low-rank blocks

When testing our code, we used double precision (FP64), constant rank 16, and a minimal block size of 256 x 256 for compression. Large low-rank blocks are partitioned deeper to at most N/8 x N/8 to increase task level parallelism. In general, without extreme tuning, we achieved satisfactory performance on a single GPU (NVIDIA Tesla V100 PCI-E 16GB), that can rival some well-optimized CPU-based hierarchical matrix libraries.

We compared to an existing $\mathcal{H}$-matrix library HLIBpro, which also implements a task-based $\mathcal{H}$-LU using only CPU. HLIBpro uses LAPACK for dense and low-rank routines. As the result shows, our implementation shows generally an order of magnitude faster than HLIBpro. HLIBPro with DAG scheduling performs worse due to the large overhead for smaller matrix sizes. It can be seen that our DAG scheduler has very little overhead, and is even negligible compared to the fast GPU execution time. Also, the scaling of our GPU $\mathcal{H}$-LU code (pspl) is very close to the ideal $\mathcal{O}(Nlog^2N)$. If we split the GPU kernel, it reduces the size of the DAG for each kernel, and we see a performance benefit when the matrix size is large. This is the first time a LU decomposition of an $\mathcal{H}$-matrix has shown $\mathcal{O}(Nlog^2N)$ complexity for a GPU implementation. Actually we are not aware of any GPU implementation of the $\mathcal{H}$-matrix LU decomposition, let alone a GPU implementation that scales ideally.

5.3 Batched randomized SVD on GPUs

The $\mathcal{H}$-matrix actually has a preparation step before it can be used for matrix mul-

tiplications and factorization. The originally dense matrix must be partitioned into a hierarchical block structure with large sub-matrices in the off-diagonal and small sub-matrices near the diagonal. Then these sub-matrices must be compressed into a low-rank matrix using U, Σ, and V as described above. The most naïve way to do this is to perform an SVD and shrink the matrices to obtain the low-rank matrices, but this is very slow. The adaptive cross approximation (ACA) samples the rows and columns of the original sub-matrix to quickly obtain a low-rank representation U and V but is known to fail for certain types of matrices. In the previous JHPCN project we developed a method to use the fast multipole method (FMM) to perform this low-rank compression. We adapted the kernel independent FMM so that the compression can be used for different equations without modifying the FMM kernels. However, this approach has a limitation that an analytical expression to generate the dense matrix is necessary. In the present fiscal year, we have investigated an alternative fast compression technique, which can be applied to any dense sub-matrix. We developed a highly optimized batched randomized SVD that can reliably compress all sub-matrices at once on the GPU, while achieving high Flop/s on the latest GPU architectures by exploiting TensorCores [5, 8].

A majority of the computation in the randomized SVD is the tall-skinny QR (TSQR) as shown in Fig. 4. Therefore, we have focused our efforts on optimizing the TSQR on modern GPU architectures with TensorCores. TSQR results in a batched QR operation, but such functions are not available in cuBLAS or MAGMA. Therefore, we implemented our own batched QR operation, which can utilize TensorCores for the GEMM operations. We use a Householder QR algorithm that utilizes the shared memory on GPUs effectively.

The results of our TSQR implementation are shown in Figs 5 and 6. The residual is

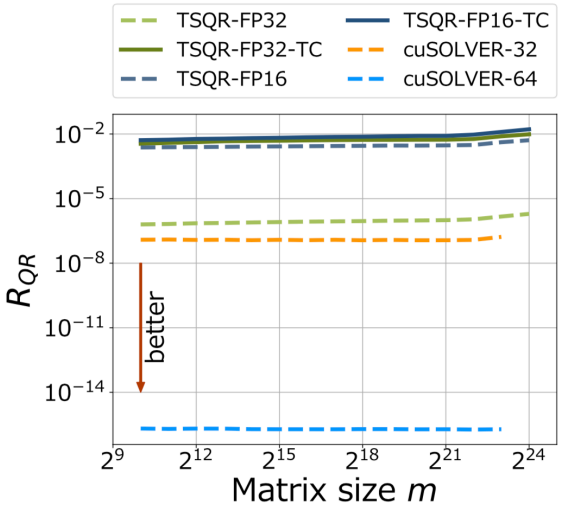

Fig. 5 Residual error of TSQR on TensorCores

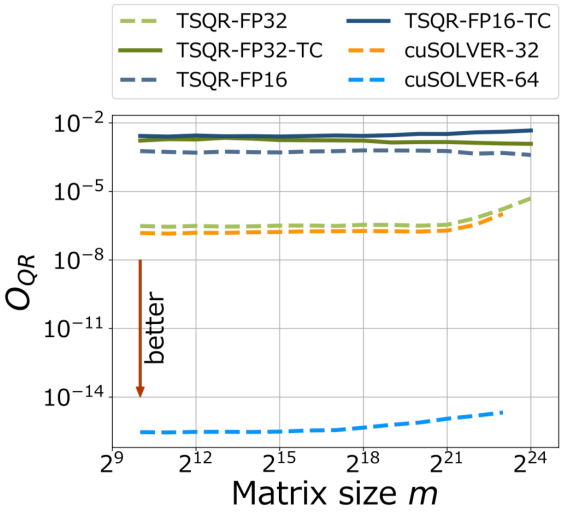

Fig. 6 Orthogonality of TSQR on TensorCores

calculated as

$$\frac{||A - QR||}{||A||}, \tag{1}$$

while the orthogonality is calculated as

$$\frac{||I - Q^T Q||}{\sqrt{N}} \tag{2}$$

We compare a FP32 version with FP16 version with and without TensorCores. The cases where TensorCores are used are noted

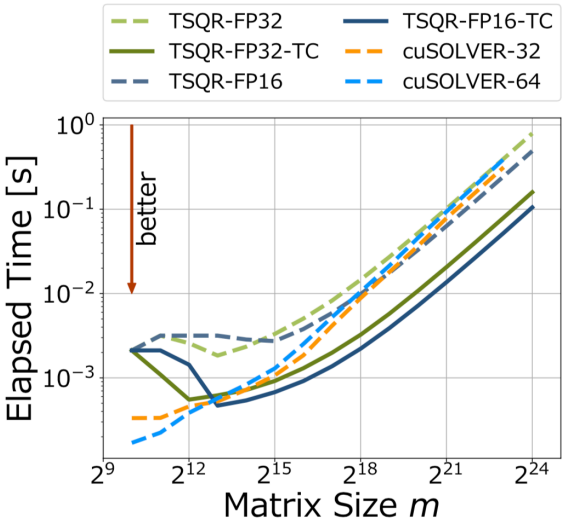

Fig. 7 Calculation time of TSQR on TensorCores

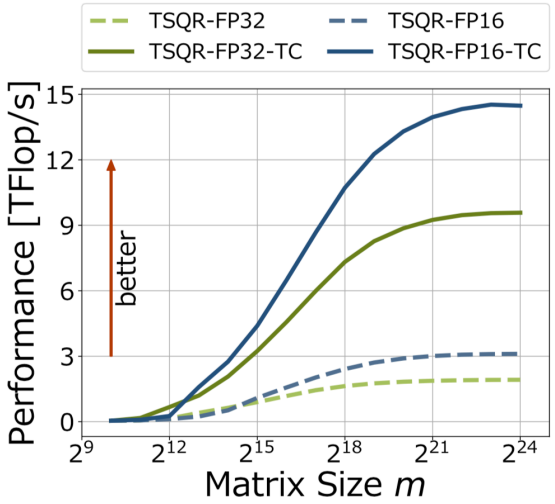

Fig. 8 Flop/s of TSQR on TensorCores

with "TC" in the legend. We also, compare with cuSOLVER with FP32 and FP64. The error of FP64 is approximately 10^{-15}, and the error of FP32 is around 10^{-6} for both our TSQR and cuSOLVER's QR. The error when using FP16 is even larger (around 10^{-2}), and this does not seem to improve when TensorCores are used. This trend is similar when looking at the orthogonality in Figure 6. TensorCores can reduce the loss of trailing digits because the accumulation is performed with FP32. However, the source of the error in this case is not the loss of trailing digits, but the truncation error when the FP32 variables are converted to FP16 during the input to TensorCores. We are currently developing a refinement procedure that can achieve close to FP32 accuracy, even when using TensorCores.

The timing results are shown in Fig. 7, while the Flop/s are shown in Fig. 8. The results using TensorCores are shown in the solid lines, while the others use a dashed line. Our TSQR using TensorCores is generally much faster than the cuSOLVER QR, except for very small matrices. We are able to achieve close to 15 TFlop/s when using TensorCores with FP16 everywhere.

5.4 Extending the $\mathcal{H}$-matrix code to QR factorization

We have already implemented the LU factorization of $\mathcal{H}$-matrices for a hybrid distributed/shared memory environment [1]. For FY2019 our goal is to extend this further to QR factorization. Before tackling the full $\mathcal{H}$-matrix QR factorization, we decided to first experiment with the QR factorization of a non-hierarchical block low-rank (BLR) matrix [2]. We developed two types of approaches for the QR factorization of BLR-matrices, i.e., a Modified Block Gram Schmidt (MBGS) algorithm-based approach and a tile-QR factorization-based approach.

5.4.1 Modified Block Gram Schmidt (MBGS) algorithm-based approach

First, we applied an approach using the modified block Gram Schmidt (MBGS) algorithm for the QR factorization of the BLR-matrices [2].

As a result QR factorization, a given matrix $A \in \mathbb{R}^{n\times n}$ is factorized as $A := QR$, where $Q \in \mathbb{R}^{n\times n}$ is an orthogonal matrix and $R \in \mathbb{R}^{n\times n}$ is an upper triangular matrix. When the matrices A, Q, R are divided into $N_b \times N_b$ blocks, we can apply the MBGS algorithm shown in Fig. 9. In Fig. 9 the subscripts of the matrices denote the row and column indices of the lattice. The symbol

```
Input : A with N_b × N_b blocks
Output : Q, R with N_b × N_b blocks
         such that QR=A
1: For   j = 1,2,⋯,N_b  do
2:     [Q_{*,j}, R_{j,j}] ≔ TSQR(A_{*,j})
3:       For   k = j + 1, j + 2,⋯,N_b  do
4:           R_{j,k} ≔ Q_{*,j}^T A_{*,k}
5:           A_{*,k} ≔ A_{*,k} − Q_{*,j}  R_{j,k}
6:       End do
7: End do
```

Fig. 9 Modified Block Gram Schmidt (MBGS) algorithm.

* represents all blocks, e.g., $A_{*,j}$ is the j-th block column which is a tall-skinny matrix. The function TSQR($A_{*,j}$) returns the QR factorization of the tall-skinny matrix $A_{*,j}$ such that $Q_{*,j}^T Q_{*,j} = I$ and $Q_{*,j}, R_{j,j} = A_{*,j}$ where $Q_{*,j} \in R^{N \times l}$ and $R_{j,j} \in R^{l \times I}$.

For the block-divided matrix, we can formally derive a BLR-matrix when all the off-diagonal sub-matrices are represented by low-rank sub-matrices. Using the MBGS algorithm in Fig. 9, we consider the QR factorization of the BLR-matrix. As with the case of QR factorization of a dense matrix, we perform the operations block-by-block. The difference is sub-matrices in the off-diagonal blocks are formatted by the low-rank representation. We enforce the following two conditions on the computed matrices Q and R,

C1: The matrices Q and R have the same lattice structure as A.

C2: The off-diagonal sub-matrices Q_{ij} and R_{ij} are formatted by the same rank k_{ij} representation as A_{ij}.

To satisfy these conditions, the operations with low-rank sub-matrices are required in addition to the usual multiplication and summation of dense matrices. As a result, the multiplication QR would not be equal to A, but an approximation. This is because the summation of the two low-rank sub-matrices

Function	# of calls	1 time complexity	Total complexity
TSQR($A_{*,j}$)	$O(N/l)$	$\tilde{o}(l^3)$	$\tilde{o}(Nl^2)$
$Q_{*,j}^T A_{*,k}$	$O((N/l)^2)$	$\tilde{o}(N + l^2)$	$\tilde{o}(N^3/l^2 + N^2)$
$A_{*,k} - Q_{*,j} R_{j,k}$	$O((N/l)^2)$	$\tilde{o}(N + l^2)$	$\tilde{o}(N^3/l^2 + N^2)$

Fig. 10 Complexity of the QR factorization of a BLR-matrix based on the MBGS algorithm

increases the rank of the resulting matrix, and the condition C2 cannot be satisfied without approximating the resulting matrix. To enforce the condition, we further recompress the resulting matrix to reduce the rank. For the approximated summation, we employ the so-called "rounded addition" method. In this method, the increased rank is reduced by the efficient use of the TSQR and singular value decomposition (SVD).

Our proposed algorithm for the QR factorization of the BLR-matrices is the algorithm which replaces the matrix operations on lines 2, 4 and 5 of MBGS (Fig. 9) with low-rank arithmetic operations. The computational complexity of our proposed algorithm depends on the block sizes. Assuming that all the blocks are square and have the same size l, the breakdown of the complexity is shown in Fig. 10. When $l \propto \sqrt{N}$, the complexity takes $\mathcal{O}(N^2)$, while it is $\mathcal{O}(N^3)$ for a dense matrix.

For the parallelization of our proposed algorithm, we use a hybrid MPI+OpenMP programming model. We assign a set of block columns to an MPI process. To balance the load among MPI processes, we adopt a block cyclic assignment strategy in the column direction. Figure 11 shows the our proposed parallel MBGS algorithm for BLR-matrices.

We here discuss the performance of our proposed algorithm. For the algorithm, we re-implemented functions of the HACApK library. As a test problem, we have selected the static electric field analyses. All calculations were carried out using an SMP cluster system, which is equipped with Intel(R) Xeon(R) E5-2680 v2 (10core × 2 sockets/node) and 32 GB DDR3 memory on

```
Input : Ã^p with N_b × N_bp blocks assigned
        to p^th MPI process
Output : Q̃^p , R̃^p with N_b × N_bp blocks
1: For j = 1,2,⋯,N_b do
2:   If(p = Mod(j−1, N_p)) then
3:       [Q̃_*j, R̃_jj] ≔ TSQR(Ã_*j) using OpenMP
4:       Broadcast(Q̃_*j, R̃_jj)
5:   End if
6:   Sync
7:   For  k = j + 1, j + 2,⋯,N_b do
8:       If(p = Mod(k − 1, N_p)) then
9:          R̃_jk ≔ Q̃*_j Ã_*k using OpenMP
10:      End if
11:  End do
12:  For all Ã_ik ∈ {Ã^p ∩ k > j} do using
     OpenMP schedule(dynamic)
13:        Ã_ik ≔ Ã_ik − Q̃_ij R̃_jk
14:  End do
15: End do
```

Fig. 11 Procedure of the parallel MBGS algorithm for BLR-matrices on the an MPI process

a node. For the interconnect, a Fat-Tree with Full-bisection bandwidth using InfiniBand FDR × 2 is used, which has a link throughput of 6.8 GB/s. We used the Intel Fortran compiler with the -O3 optimization option and the Intel MPI and MKL libraries.

Our proposed MBGS algorithm provides the approximation of QR factorization of the BLR-matrix. In addition to the error, the orthogonality of the resulting matrix is an important index to evaluate the quality of the QR factorization. We investigate the accuracy of the error and the orthogonality of Q and observe dependency on matrix size. Figure 12 shows the results. The error and the accuracy of orthogonality are the same order as the accuracy of the BLR-matrix to be factorized.

Figure 13 shows the parallel scalability using the hybrid MPI+OpenMP on the distributed memory system and the computational time when varying the number of cores from 1 to 200. In the calculations, we use a single MPI process with 10 OpenMP threads per socket, which means two MPI processes per a node. Parallel speed-up up to about 140 cores is observed in Fig. 13 (a). The fastest time is about 20-fold faster than the time of serial computation. As Fig.13 (b) shows, similar results are observed if we use test matrices of different sizes.

5.4.2 Tile-QR factorization-based approach

We extend the tile-QR factorization to handle low-rank off-diagonal blocks.

A graphical representation of the tile-QR factorization of a BLR matrix is shown in Fig. 14. Each picture shows a LAPACK subroutine. `GEQRT` performs the QR decomposition where

$$QR(A_{kk}) \to (Y_{kk}, T_{kk}, R_{kk}). \tag{3}$$

`LARFB` applies the orthogonal matrix to obtain the R matrix as

$$R_{kj} = (I - Y_{kk} T_{kk}^T Y_{kk}^T) A_{kj}. \tag{4}$$

`TPQRT` performs a block QR decomposition where

$$QR \begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix} \to (Y_{ik}, T_{ik}, R_{kk}). \tag{5}$$

`TPMQRT` applies the block orthogonal matrix to obtain the block R matrix as

$$\begin{pmatrix} R_{kj} \\ A_{ij} \end{pmatrix} = \left(I - \begin{pmatrix} I \\ Y_{ik} \end{pmatrix} T_{ik}^T \left(I Y_{ik}^T \right) \right) \begin{pmatrix} R_{kj} \\ A_{ij} \end{pmatrix} \tag{6}$$

What is different from the existing work is the fact that we have low-rank blocks on the off-diagonals during the tile-QR factorization, which requires special care if one wishes to keep the low-rank blocks from becoming dense during the factorization.

Similar to the TSQR case, we show the residual error in Fig. 15 and the orthogonality in 16. We are using FP64 for all calculations in this case. The large residual error is coming from the low-rank approximation of the BLR matrix. The orthogonality seems to be quite good even though the residual error is fairly large.

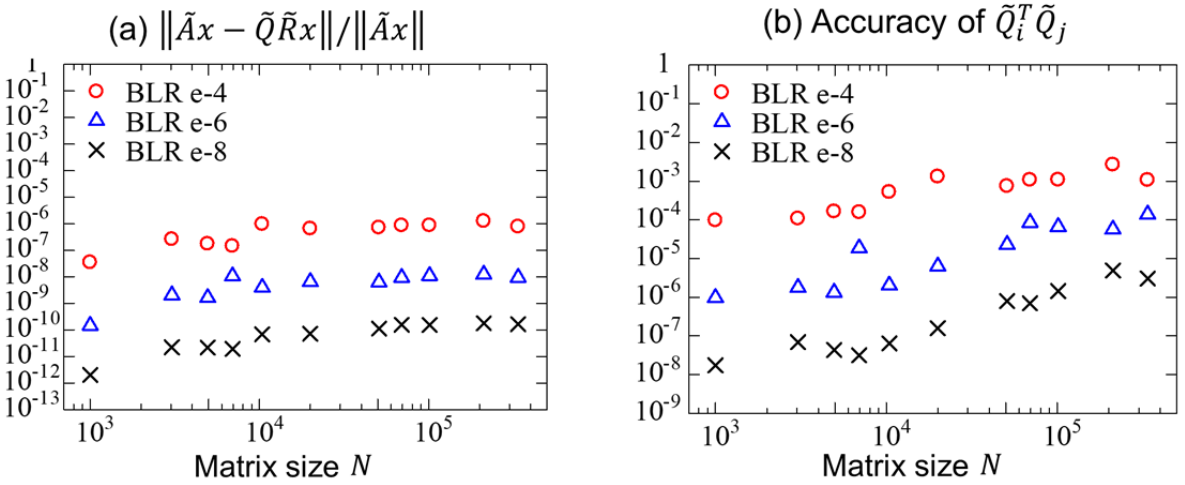

Fig. 12 The error of factorization (a) and accuracy of orthogonality of the resulting orthogonal matrix Q (b) derived from our proposed MBGS algorithm for BLR-matrices.

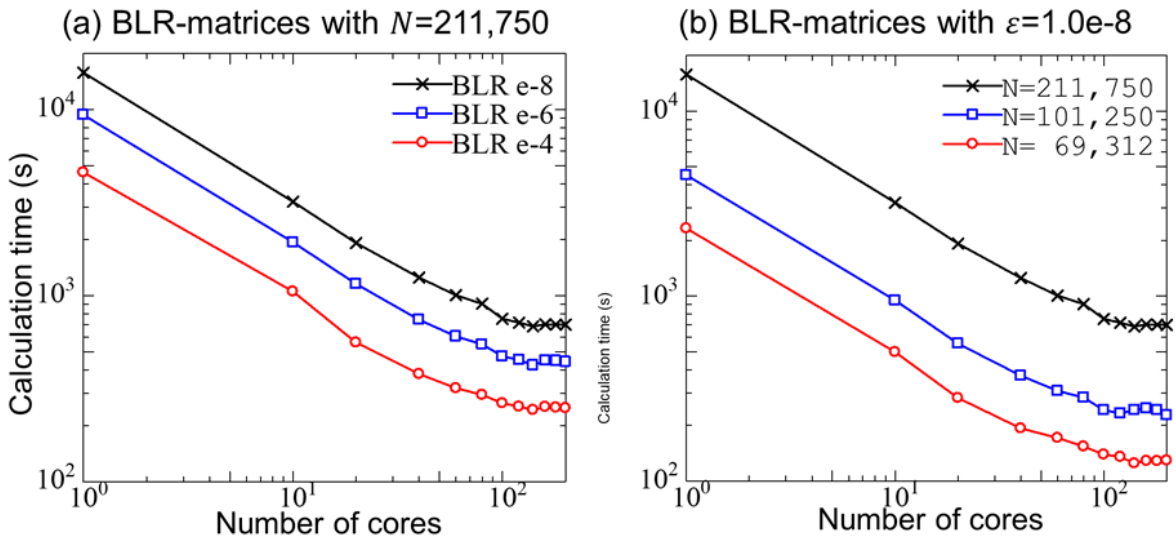

Fig. 13 Parallel scalability of parallel MBGS for BLR-matrices.

6 Progress during FY2019 and Future Prospects

We were able to achieve all four project goals in our proposal for FY2019. The implementation of the nested basis version is now complete, and we are investigating how to parallelize this efficiently on multiple-GPUs using task-based runtime systems. The use of runtime for $\mathcal{H}$-LU on GPU is complete and has been presented as a poster at SC'19 [7]. Batched randomized SVD on GPUs has been implemented using TensorCores with mixed precision and presented as a poster at SC'19 (best poster candidate) [5,8]. Extending the $\mathcal{H}$-matrix code to QR factorization has been implemented using Modified Gram-Schmidt [2] and a block Householder transformation. The results have been compared to show superior complexity for the Gram-Schmidt,

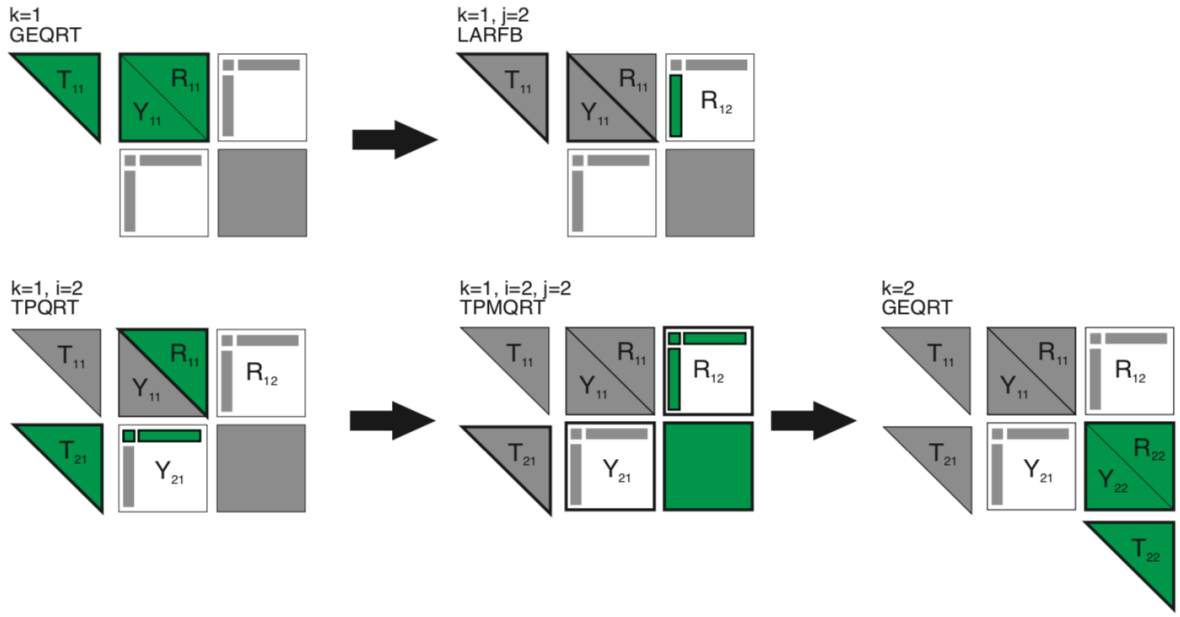

Fig. 14 Graphical representation of the QR factorization of a BLR matrix

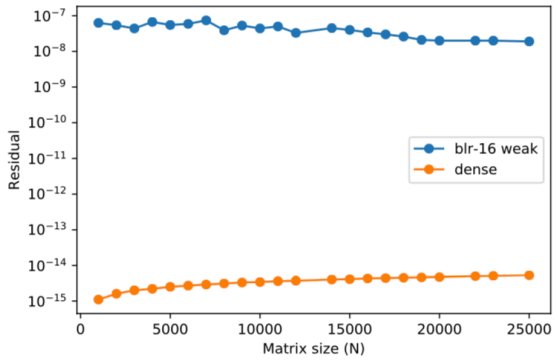

Fig. 15 Residual error of BLR QR

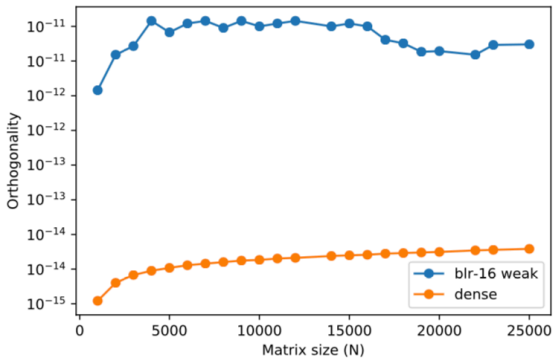

Fig. 16 Orthogonality of BLR QR

while the parallel efficiency and orthogonality of the resulting $\mathcal{H}$-QR factorization was superior for the Householder transformation.

We plan to further enhance the capability of our generalized hierarchical low-rank approximation code. Now that a highly optimized TSQR on GPU is available, its extension to randomized SVD or interpolatory decomposition (ID) is straightforward. The reason for this is because the hotspot in these methods is thee TSQR. The extension of BLR-QR to H-QR is almost completed. The code is written and is going through some final debugging for corner cases.

7 List of Publications and Presentations

Journal Papers (Refereed)

1. I. Yamazaki, A. Ida, R. Yokota, Jack Dongarra, Distributed Memory Lattice H-matrix Factorization, The International Journal of High Performance Computing Applications (2019).
2. A. Ida, H. Nakashima, T. Hiraishi,

I. Yamazaki, R. Yokota, and T. Iwashita, QR Factorization of Block Low-Rank Matrices with Weak Admissibility Condition, IPSJ Trans. ACS (2019).

3. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, M. Yasugi, Parallelization of Matrix Partitioning in Construction of Hierarchical Matrices using Task Parallel Languages, IPSJ Trans.ACS (2019). (Outstanding Research Award).
4. M. Abduljabbar, M. Al Farhan, N. Al-Harthi, R.Chen, R.Yokota, H. Bagci, D. Keyes, Extreme Scale FMM-Accelerated Boundary Integral Equation Solver for Wave Scattering, SIAM Journal on Scientific Computing, Vol. 41, No. 3, pp. C245–C268 (2019).

Proceedings of International Conferences (Refereed)

5. H. Ootomo, R. Yokota, Batched QR Decomposition Using TensorCores, The 81st National Convention of IPSJ, Fukuoka, Japan, March 14-16, 2019.
6. S. Ohshima, I. Yamazaki, A. Ida, R. Yokota, Optimization of Numerous Small Dense-Matrix-Vector Multiplications in H-matrix Arithmetic on GPU, ATMG In IEEE MCSoC, Singapore, October 1, 2019.
7. Q. Ma, R. Yokota, Runtime System for GPU-based Hierarchical LU Factorization, SC19 research poster, Denver, Colorado, 17-22 November, 2019.
8. H. Ootomo, R. Yokota, TSQR on TensorCores, SC19 research poster, Denver, Colorado, 17-22 November, 2019. (best poster candidate).
9. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, M. Yasugi, Implementation of Partitioning of Hierarchical Matrices using Task Parallel Languages, ICPP 2019, (Best Poster Award).

Proceedings of International Conferences (Non-refereed)

Presentations at domestic conference (Non-refereed)

10. P. Spalthoff, R. Yokota, Flexible and Simplistic Hierarchical Matrix-Based Fast Direct Solver, The 170th Workshop on High Performance Computing (SWoPP2019), Kitami, Japan, July 24, 2019.

Other (patents, press releases, books and so on)