SLATE performance improvements: QR and eigenvalues

Kadir Akbudak
Paul Bagwell
Sebastien Cayrols
Mark Gates
Dalal Sukkari
Asim YarKhan
Jack Dongarra

Innovative Computing Laboratory

April 19, 2021

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Revision	Notes
04-2021	first publication

```
@techreport{akbudak2021slate-qr-eig,
  author={Kadir Akbudak and Paul Bagwell and Sebastien Cayrols and Mark Gates
          and Dalal Sukkari and Asim YarKhan and Jack Dongarra},
  title={{SLATE} performance improvements: {QR} and eigenvalues, {SWAN} No. 17},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2021},
  month={4},
  number={ICL-UT-XX-XX},
  note={revision 04-2021},
  url={https://www.icl.utk.edu/publications/swan-017},
}
```

Contents

1 Introduction

SLATE (Software for Linear Algebra Targeting Exascale) [1] is being developed as part of the Exascale Computing Project (ECP) [2], which is a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration (NNSA). The objective of SLATE is to provide distributed, GPU-accelerated dense linear algebra capabilities to the US Department of Energy and to the high-performance computing (HPC) community at large.

This report will discuss current efforts in improving performance in SLATE, with focus on the QR and eigenvalue factorizations. These improvements are intended to be general and many of them should be applicable to the other algorithms implemented in SLATE.

2 QR factorization (geqrf)

The Communication Avoiding QR (CAQR) factorization algorithm used in the geqrf operation is presented in the SLATE Working Note 7 [1]. In this section, we present two performance enhancements to the geqrf operation in SLATE.

2.1 GPU Implementation of Triangular Matrix-Matrix Multiplication (trmm)

The unmqr operation used in geqrf heavily depends on trmm and gemm operations. The GPU implementation of gemm is used during unmqr, however the GPU implementation of trmm was missing. Batched trmm operation is implemented for GPUs.

Figure 1 presents the performances of the new GPU implementation of the trmm operation, as well as the existing GPU implementation of gemm and CPU implementation of trmm. The experiments are performed on a DGX-1 system with eight V100 SXM2 GPUs. The system has two Intel Xeon CPUs (E5-2698 v4 @ 2.20GHz with 20 cores). Each GPU can deliver 7.8 TFlop/s when double precision arithmetic is used. The number of OpenMP threads is set to 40 for the trmm runs on the host, otherwise the number of OpenMP threads is set to 20. All GPU runs are performed on a single GPU. The result for the best performing tile size is reported. As seen in Figure 1, gemm achieves 7 TFlop/s, which is 90% of the peak performance. The performance of the new GPU implementation of trmm is very close to that of gemm.

2.2 GPU Implementation of unmqr

The QR factorization implementation uses the unmqr operations to apply Householder reflectors from the local QR factorization during the trailing matrix update. The only missing GPU kernel within unmqr was trmm. So after the new GPU implementation of trmm presented in Section 2.1, we implement a new GPU version of unmqr. The new implementation is traced in order to discover an unnecessary data movements between the host and the GPU. Only a copy operation is detected causing extra data movement and it is fixed.

[1] http://icl.utk.edu/slate/
[2] https://www.exascaleproject.org

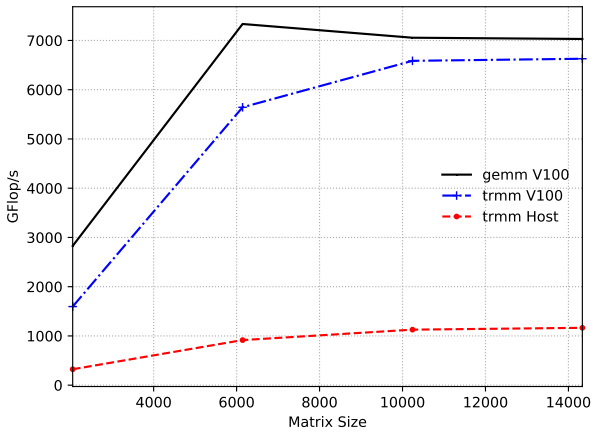


Figure 1: Performance of gemm and trmm in terms of GFlop/s on the host and one V100. The performance of the new GPU implementation of trmm is very close to that of gemm.

Figure 2 presents the performance of the geqrf operation before and after the addition of the GPU implementation of the unmqr operation. The experiments are performed on the same system mentioned in Section 2.1. As seen in the figure, there is considerable performance improvement for larger matrix sizes when the GPU kernel for unmqr is used within the geqrf operation in SLATE.

We identified additional improvements that can be made in QR, such as moving look-ahead panel updates to the GPU. We are in the process of implementing these additional changes, which we expect to provide significant performance improvement.

3 Hermitian eigenvalue problem (heev and syev)

The Hermitian eigenvalue computations are essential for many scientific problems, such as, quantum chemistry, quantum physics. Solving the Hermitian eigenvalue problem is to decompose the dense matrix $A \in \mathbb{R}^{n \times n}$ into $A = V \Lambda V^{\top}$, where, $\Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_n)$ is the matrix of all the eigenvalues, and V is orthogonal matrix containing the eigenvectors ($VV^{\top} = I$).

3.1 Three Stage Algorithms

We solve the symmetric eigenvalue problem by a three stage algorithm, shown in Figure 3: The three stages algorithm to solve the Hermitian eigenvalue problem is presented in SLATE Working Note 13 [2]. To ensure this report is self-contained, we briefly recall these stages:

(1) First stage reduction from full to symmetric band (eigenvalue) form, which uses Level 3 BLAS.

(2) Second stage reduction band to symmetric tridiagonal (eigenvalue) form. This uses a bulge chasing algorithm.

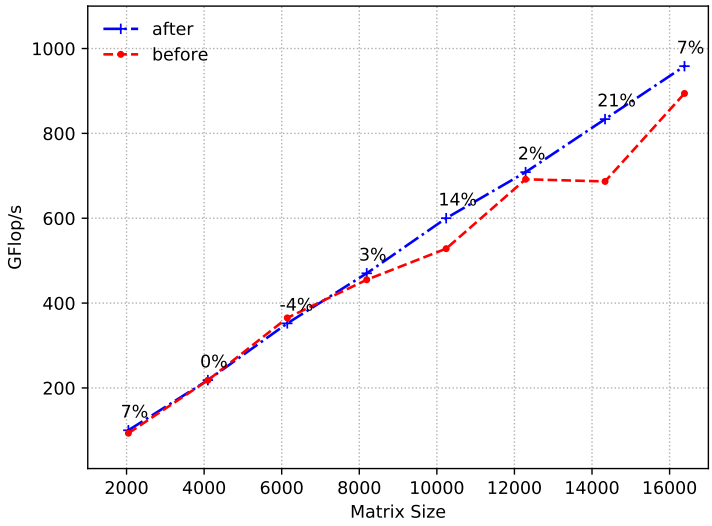


Figure 2: Performance of the geqrf operation before and after the addition of the new GPU implementation of unmqr. With the new unmqr GPU kernel, 7% improvement is observed in the performance of geqrf for the largest matrix.

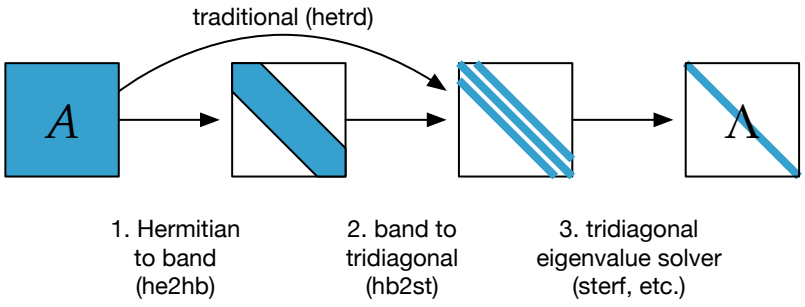


Figure 3: Three stage symmetric/Hermitian eigenvalue and SVD algorithms.
Three stage symmetric/Hermitian eigenvalue (top) and SVD (bottom) algorithms.

(3) Third stage reduction to diagonal form, revealing the eigenvalues. Currently we use QR iteration, but could also use divide and conquer, MRRR, bisection, or other solver.

This is in contrast to the traditional algorithm used in LAPACK and ScaLAPACK that goes directly from full to bidiagonal or symmetric tridiagonal, which uses Level 2 BLAS and is memory-bandwidth limited.

The first stage proceeds by computing a QR factorization of a block column to annihilate entries below the diagonal, and updating the trailing matrix. It repeats factoring block columns, until the entire matrix is brought to band form. The width of the block columns and rows is the resulting matrix bandwidth, n_b.

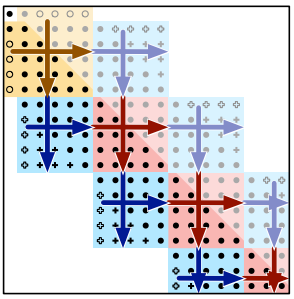


Figure 4: Symmetric bulge-chasing algorithm. Only the lower triangle is accessed; the upper triangle is known implicitly by symmetry.

The second stage reduces the band form to the final tridiagonal form using a bulge chasing technique, as shown in Figure 4. Symmetry is taken into account, so only entries in the lower triangle are computed, while entries in the upper triangle are known by symmetry. It involves $6n_bn^2$ operations, so it takes a small percentage of the total operations, which decreases with n. The operations are memory bound, but are fused together as Level 2.5 BLAS [3] for cache efficiency. We designed the algorithm to use fine-grained, memory-aware tasks in an out-of-order, data-flow task-scheduling technique that enhances data locality [4, 5].

The second stage proceeds in a series of sweeps, each sweep bringing one row to tridiagonal and chasing the created fill-in elements down to the bottom right of the matrix using successive orthogonal transformations.

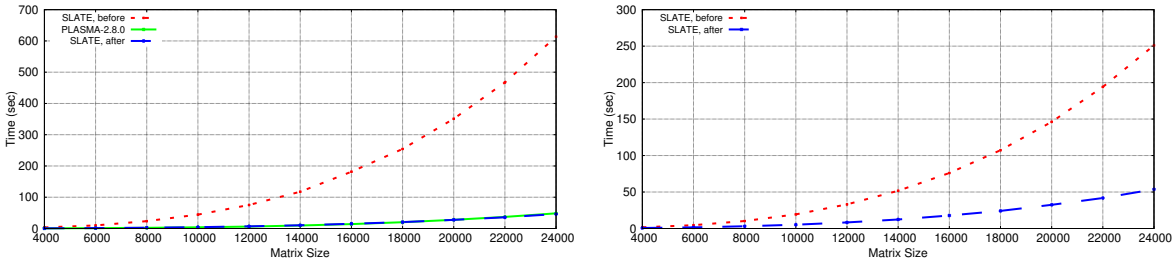
Once the tridiagonal reduction is achieved, the implicit QR eigensolver `steqr2` calculates the eigenvalues and optionally its associated eigenvectors of the condensed matrix structure. The `steqr2` is a modified version of LAPACK routine `steqr`, to allow each process to perform updates on the distributed matrix Q_2, and achieve some parallelization during this step.

3.2 Performance Improvements

We optimized the performance of Hermitian to Hermitian band reduction `he2hb`, by looking at the dependency graph and insert OpenMP tasks to improve and extend the parallelization of the various operations of `he2hb`. Algorithm 3.1 presents the `he2hb` implementation and parallelization using OpenMP tasks.

Algorithm 3.1 Hermitian to Hermitian band reduction (he2hb).

```
for k = 0 to nt − 1 do
    // QR of panel
    internal::geqrf<Target::HostTask>(A_panel, Tlocal_panel, ib, max_panel_threads);
    internal::ttqrt<Target::HostTask>(A_panel, Treduce_panel);
    // QR update trailing submatrix
    for i = k + 1 to nt − 1 do
        #pragma omp task depend(inout:row[i])
        for j: indices do
            if i == j then
                hemm(Side::Left, 1.0, A(i, j), A(j, k), 1.0, W(i, k));
            else if i > j then
                gemm(1.0, A(i, j), A(j, k), 1.0, W(i, k));
            else
                gemm(1.0, conj_transpose(A(j, i)), A(j, k), 1.0, W(i, k));
            end if
        end for
        i0 = indices[0];
        auto TVAVT0 = W.sub(i, i, k, k);
        auto T0 = Tlocal.sub(i0, i0, k, k);
        if T0.mb < T0.nb then
            // trapezoid
            T0 = T0.slice(0, mb-1, 0, mb-1);
            TVAVT0 = TVAVT0.slice(0, mb-1, 0, nb-1);
        end if
        auto Tk0 = TriangularMatrix(Uplo::Upper, Diag::NonUnit, T0);
        trmm(Side::Right, Diag::NonUnit, 1.0, Tk0(0, 0), TVAVT0(0, 0));
    end for
    if A.tileIsLocal(i0, i0) then
        auto TVAVT = W(0, 0);
        for i: indices do
            #pragma omp task depend(in:row[i]) depend(inout:block[0])
            gemm(1.0, conj_transpose(A(i, k)), W(i, k), 1.0, TVAVT);
        end for
        auto TVAVT0 = W.sub(0, 0, 0, 0);
        auto T0 = Tlocal.sub(i0, i0, k, k);
        if T0.mb < T0.nb then
            // trapezoid
            T0 = T0.slice(0, mb-1, 0, mb-1);
            TVAVT0 = TVAVT0.slice(0, mb-1, 0, nb-1);
        end if
        auto Tk0 = TriangularMatrix(Uplo::Upper, Diag::NonUnit, T0);
        #pragma omp task depend(in:block[k]) depend(inout:block[0])
        trmm(Side::Left, Diag::NonUnit, 1.0, conj_transpose(Tk0(0, 0)), TVAVT0(0, 0));
        for i: indices do
            #pragma omp task depend(in:block[0]) depend(inout:row[i])
            gemm(-0.5, A(i, k), TVAVT, 1.0, W(i, k));
        end for
        #pragma omp taskwait
        internal::her2k<Target::HostTask>(-1.0, A.sub(k+1, nt-1, k, k), W.sub(k+1, nt-1, k, k), 1.0, A.sub(k+1, nt-1));
    else
        for j = k + 1 to nt − 1 do
            #pragma omp task depend(in:row[j]) depend(in:block[k]) depend(inout:block[j])
            for i: indices do
                if i > j then
                    gemm(-1.0, A(i, k), conj_transpose(W(j, k)), 1.0, A(i, j));
                else
                    gemm(-1.0, W(j, k), conj_transpose(A(i, k)), 1.0, A(j, i));
                end if
            end for
        end for
    end if
    internal::hettmqr<Target::HostTask>( Op::ConjTrans, A_panel, Treduce_panel, A.sub(k+1, nt-1));
end for
```



(a) 1×1 grid configuration

(b) 2×2 grid configuration

Figure 5: Performance comparison of Hermitian to Hermitian band reduction.

We studied the impact of inserting OpenMP tasks on a single node on a local machine Saturn using CPU only. The performance optimization using GPUs is an ongoing effort. The effect of our performance updates can be seen in the performance of the double-precision `he2hb` shown in Figure 5. Figure 5a compares the performance of `he2hb` before and after parallelizing the code using OpenMP tasks against the PLASMA implementation of `he2hb`. Inserting OpenMP tasks achieves up to 10× speedup compared to its original implementation. Moreover, SLATE and PLASMA `he2hb` implementations now perform similarly. Figure 5b shows up to 4× improvement in the `he2hb` performance using a 2×2 grid configuration.

4 Triangular solves (trsm)

Many operations such as LU factorization require at some point to solve multiple triangular systems, either with one or more *right-hand sides* (RHS):

$$op(A)X = B, \tag{1}$$

where $op(A) = A$ or A^H is either an upper or a lower triangular matrix of dimension $m \times m$, B is a matrix that contains n RHS of dimension m, and X is the requested solution with the same dimension as B.

In SLATE, the matrices A, B, and X are stored using tiles of size nb. Figure 6 shows the representation of the data in the case of the forward substitution, i.e., A is a lower triangular matrix. For distributed computation, tiles are distributed among the processors and thus they have to be communicated when and where they are needed. For example, the first step during the forward substitution is the computation of the solution of $A_{1,1}X_1 = B_1$. If $A_{1,1}$ and B_1 are not located on the same processor, we have to move either $A_{1,1}$ where B_1 is or vice versa.

4.1 trsm implementation

The existing trsm in SLATE targets solving many triangular systems (right-hand sides) involving the same matrix A, i.e., n is large. Therefore, the implemented strategy is to move tiles of A to where the associated tiles of B are located, and is detailed in Algorithm 4.1. For simplicity, we assume that the dimensions of the matrices are multiples of nb. Therefore we have $m = k * nb$

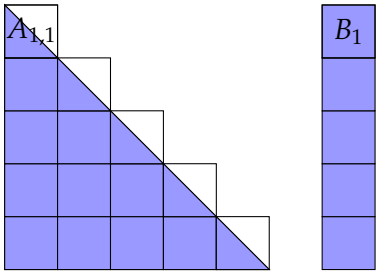


Figure 6: Data representation of Equation (1), where the computation is done inplace (colored area represent the actual data while the shape represents the storage in memory).

and $n = l * nb$. Moreover, we deliberately omit the update of B as it is not relevant regarding the data movement. The algorithm iterates over the k diagonal tiles of A. At iteration i, we move $A_{i,i}$ to where the corresponding tiles of B, $B_{i,:}$ are located. Then the computed solution, $B_{i,:}$ is gathered along with the off diagonal tiles $A_{i+1:k,i}$ where the sub-tiles of $B_{i,:}$ are located. Note that we use the MATLAB notation. Therefore, $B_{i,:} = B_{i,1:l}$ is the i-th block row of the matrix B.

Algorithm 4.1 trsm(A,B)

Require: $A \in \mathbb{C}^{m \times m}$
Require: $B \in \mathbb{C}^{m \times n}$
  nb is the tile size
  $m = k * nb$
  $n = l * nb$
  for i=1:k **do**
    Broadcast $A_{i,i}$ where $B_{i,:}$ are located,
    Solve inplace $A_{i,i}X_{i,:} = B_{i,:}$,
    for j=i+1:k **do**
      Broadcast $A_{j,i}$ where $B_{j,:}$ are located,
    end for
    for j=1:l **do**
      Broadcast $B_{i,j}$ where $B_{i+1:k,j}$ are located,
    end for
  end for

We now compute the communication cost of this strategy. A simple approach to evaluate the cost is the case where the number of RHS is fewer than or equal to the tile size, i.e., $n \leq nb$ and so $l = 1$. Based on it and used in the proof, we present the more general case in Lemma 4.1.

Lemma 4.1. *Assume that A is a triangular matrix of dimension $m \times m$ and B is a matrix of dimension $m \times n$, both distributed as tiles of size nb. Assume that $m = k * nb$ and $n = l * nb$. Finally, assume that tiles are distributed such that a processor owns at most one tile of A and B.*

Let c_A be the cost to move one tile of A, and c_B be the cost to move one tile of B.

The overall cost C of Algorithm 4.1 is given by

$$C = c_A \log(l)\,(2k-1) + c_B \log((k-1)!)\,. \tag{2}$$

Proof. In order to prove Equation (2), we first consider the simplest case, we then extend the result. We also consider the case that maximizes the communication which is a process owns a single tile of A and B.

Assume that A is a triangular matrix of dimension $m \times m$ and B is a matrix of dimension $m \times n$, both distributed as tiles of size nb. Assume that $m = k * nb$ and $n \leq nb$, i.e., $l = 1$.

Let c_A be the cost to move one tile of A, and c_B be the cost to move one tile of B.

The cost C to solve Equation (1) using Algorithm 4.1 is given by

$$C = \sum_{i=1}^{k-1} (c_A + c_A + c_B \log(k-i)) + c_A \tag{3}$$
$$= 2c_A (k-1) + c_B \log((k-1)!) + c_A \tag{4}$$
$$= c_A (2k-1) + c_B \log((k-1)!). \tag{5}$$

We now extend the case to any l. The main difference comes from the broadcast that adds the term $\log(l)$ to each data movement that involves a tile of A. Equation (3) becomes:

$$C = \sum_{i=1}^{k-1} (c_A \log(l) + c_A \log(l) + c_B \log(k-i)) + c_A \log(l) \tag{6}$$
$$= c_A \log(l) (2k-1) + c_B \log((k-1)!). \tag{7}$$

□

For a small number of RHS, say $l = 1$, we have $c_A \gg c_B$. Moreover, for a reasonable number of tiles k, Equation (5) is dominated by the term $c_A (2k-1)$. We thus propose implementing a variant where tiles of B are moved instead of tiles of A. We refer to this variant as *trsmA*.

4.2 trsmA: a variant of trsm targeting a small number of RHS

In this section, we present the implementation of the trsmA and compare it with trsm to show its limitation.

We first present in Algorithm 4.2 the modification that allows us to move the tiles of B instead of the tiles of A. Since the tiles of B are moved where the associated tiles of A are located, it implies that the contributions are also distributed. Therefore, in order to solve $A_{i,i}X_{i,:} = B_{i,:}$, we need to perform a reduction operation of the form $\sum_{j=1}^{i} B_{i,:}^{(j)}$ where $A_{i,i}$ is located. Then we solve the associated system before we broadcast the solution $B_{i,:}$. We also need to send back the tiles of this solution where they are originally located.

Similarly to the trsm implementation, we can compute the communication cost of the trsmA algorithm.

Lemma 4.2. *Assume that A is a triangular matrix of dimension $m \times m$ and B is a matrix of dimension $m \times n$, both distributed as tiles of size nb. Assume that $m = k * nb$ and $n = l * nb$. Assume that tiles are distributed such that a processor owns at most one tile of A and B.*

Algorithm 4.2 trsmA(A,B)

Require: $A \in \mathbb{C}^{m \times m}$
Require: $B \in \mathbb{C}^{m \times n}$
  nb is the tile size
  $m = k * nb$
  $n = l * nb$
  for i=1:k **do**
    Reduce contribution $B_{i,:}^{(j)}$ where $A_{i,i}$,
    Solve inplace $A_{i,i} X_{i,:} = B_{i,:}$,
    Broadcast $B_{i,:}$ where $A_{i+1:k,i}$ are located,
    Send back $B_{i,:}$ where it should be located,
  end for

Let c_B be the cost to move one tile of B.

The overall cost C of Algorithm 4.2 is given by

$$C = l c_B \times \frac{(k+3)\,k}{2} + l c_B \log\left((k-1)!\right). \tag{8}$$

Proof. In order to prove Equation (8), we first consider the simplest case, then we extend the result. We also consider the case that maximizes the communication which is a process owns a single tile of A and B.

Assume that A is a triangular matrix of dimension $m \times m$ and B is a matrix of dimension $m \times n$, both distributed as tiles of size nb. Assume that $m = k * nb$ and $n \leq nb$, i.e., $l = 1$.

Let c_B be the cost to move one tile of B.

The cost C to solve Equation (1) using Algorithm 4.2 is given by

$$C = c_B + c_B \log(k-1) + c_B + \sum_{i=2}^{k-1} \left((i+1)\,c_B + c_B \log(k-i)\right) + k c_B + c_B \tag{9}$$

$$= c_B\,(k+3) + c_B \log(k-1) + c_B \left(\frac{(k+3)\,(k-2)}{2} + \log\left((k-2)!\right) \right) \tag{10}$$

$$= c_B \times \frac{(k+3)\,k}{2} + c_B \log\left((k-1)!\right). \tag{11}$$

We now extend the case to any l. This means c_B is replaced by $l c_B$. Equation (9) becomes:

$$C = l c_B \times \frac{(k+3)\,k}{2} + l c_B \log\left((k-1)!\right). \tag{12}$$

□

4.3 Performance comparison

The trsmA algorithm aims to reduce the execution time for a small number of RHS. We now study the limit of this variant in the extreme case of one RHS and compare its cost with the

trsm algorithm. We want to determine whether the trsmA implementation is always better in this extreme case.

We first recall the difference of cost for each approach: $c_A\,(2k-1)$ and $c_B \times \frac{(k+3)k}{2}$ for moving A, B respectively. For this comparison, we also have $c_A = nb * c_B$. This gives us:

$$\Delta C = nb * c_B\,(2k-1) - c_B \times \frac{(k+3)\,k}{2} \tag{13}$$

$$= \frac{c_B}{2}\left(-k^2 + k\,(4nb-3) - 2nb\right). \tag{14}$$

Equation (14) is equal to 0 for a value of k greater than $4nb$.

The trsmA variant has therefore some limitation, mainly coming from the reduce operation in Algorithm 4.2 that is linear. Therefore, this variant may not be beneficial for a large number of tiles. However, in practice, we are not in the case where each process owns a single tile of A and B.

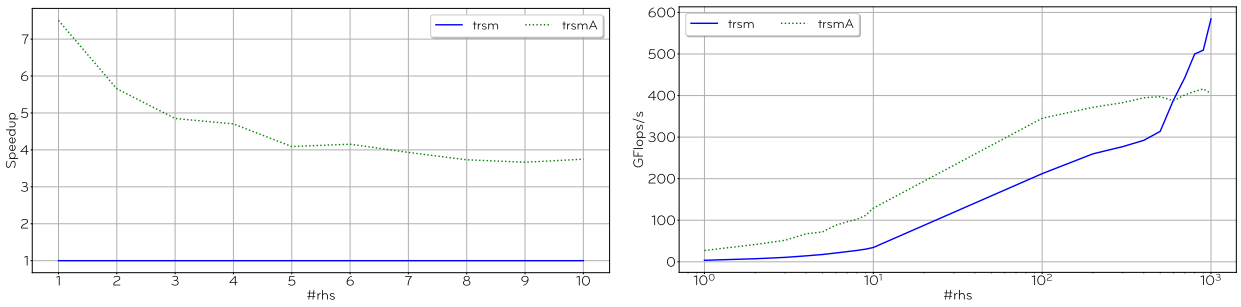
4.4 Experimental results

In this section, we compare the trsmA with the original trsm on the Summit supercomputer. We focus our study on CPU only. The GPU version is an ongoing work. We do all our experiments on four nodes, 24 MPI ranks, and 7 cores per rank, giving a grid size of 4×6. We use the following software stack: spectrum-mpi/10.3.1.2-20200121, essl/6.2.1-0, netlib-lapack/3.8.0, and netlib-scalapack/2.0.2.

We first study the performance for different numbers of RHS, in Figure 7. We take as example a matrix of size 50 000 and a tile size of 448 since this value gives the best performance for Cholesky. In Figure 7a, the number of RHS varies from one to 10. We observe that we reach up to 7.5× speedup for one RHS. When the number of RHS increases, we see less performance improvement, reaching around 4× speedup. In Figure 7b, we keep on increasing the number of RHS. We note that for #RHS equal to 600, both algorithms perform at the same speed. For more RHS, the number of RHS is too large to allow the trsmA approach to be beneficial. It actually make sense since $c_A = c_B$ in that case.

Next, we focus our study on a single RHS. Figure 8 shows the impact of the tile size for different matrix size on the performance. We take as the reference the runtime of trsm on a matrix of size 100 000 and we compute the speedup by comparing trsmA with trsm. Results confirm the interest of using this variant. The larger the dimension of the matrix is, the more the speedup we obtain. For the largest dimension, 100 000, we reach a 12.6× speedup for $nb = 768$. Note that the best speedup is not always given by the same tile size. The trend seems to show that increasing the dimension of A may require a larger tile size.

To highlight the benefit of trsmA over trsm, we recorded execution trace for each method and we show them in Figure 9. In this example, the matrix is of dimension 50 000, the tile size of 448, and only one RHS. As we suspected, we observe that a large portion of the execution is spent in waiting the tiles of A, in Figure 9a. On the other hand, this waiting time is drastically reduced, as shown in Figure 9b.



(a) Speedup for small #RHS **(b)** Performance for larger #RHS

Figure 7: Performance comparison between trsm and trsmA on a matrix of dimension 50 000 on four nodes, 24 MPI ranks, a tile size $nb = 448$.

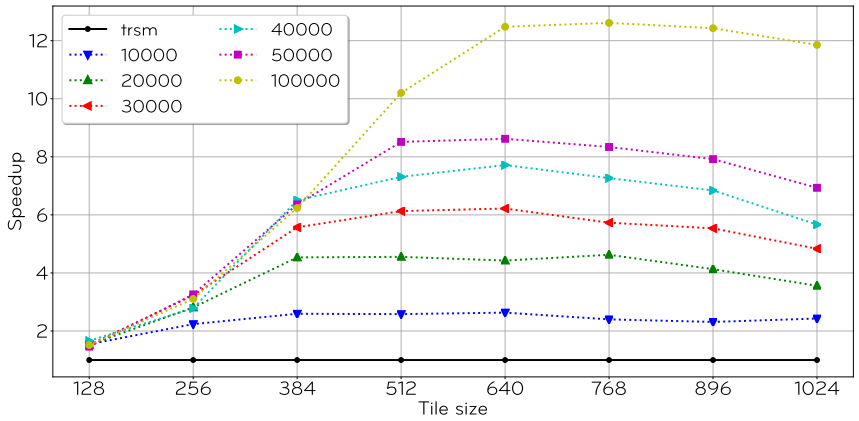
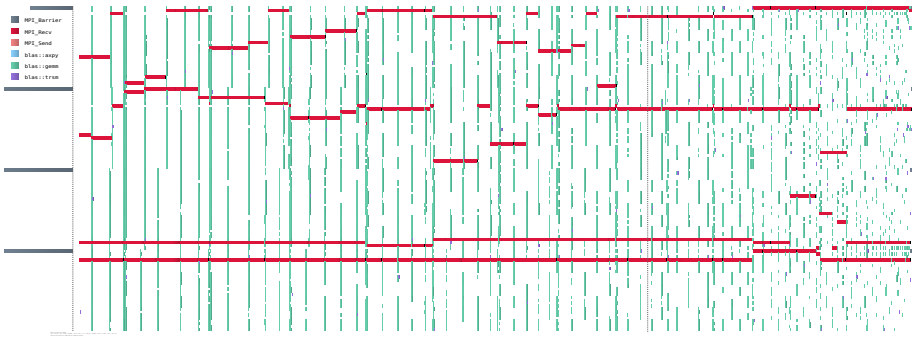
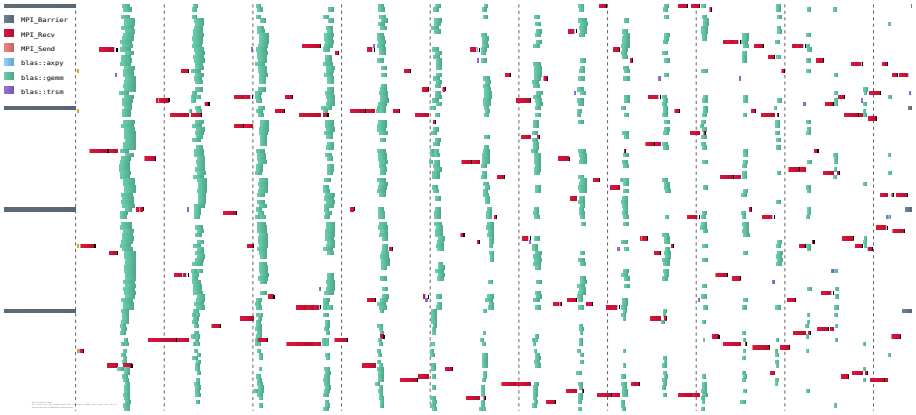


Figure 8: Performance comparison between trsm and trsmA on different matrix size and for different tile size, on four nodes, 24 MPI ranks.



(a) trsm



(b) trsmA

Figure 9: Comparison of the trace for both trsm and trsmA on a matrix of dimension 50 000 on four nodes, 24 MPI ranks, a tile size $nb = 448$.

We have presented trsmA, a variant of the already implemented trsm, where the tiles of B are moved instead of the tiles of A. This approach gives SLATE great improvement for a small number of RHS, reaching $12\times$ speedup in the case of a matrix A of dimension 100 000. However, it remains to obtain a better bound that will tell us which variant to use, depending on the grid configuration for example. We also expect this approach to be more beneficial in the GPU case.

References

[1] Mark Gates, Ali Charara, Jakub Kurzak, Asim YarKhan, Ichitaro Yamazaki, and Jack Dongarra. SLATE working note 9: Least squares performance report. Technical Report ICL-UT-11-28, Innovative Computing Laboratory, University of Tennessee, December 2018. revision 12-2018.

[2] Mark Gates, Mohammed Al Farhan, Ali Charara, Jakub Kurzak, Dalal Sukkari, Asim YarKhan, and Jack Dongarra. SLATE working note 13: Implementing singular value and symmetric/hermitian eigenvalue solvers. Technical Report ICL-UT-19-07, Innovative Computing Laboratory, University of Tennessee, September 2019. revision 04-2020.

[3] Gary W Howell, James W Demmel, Charles T Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):14, 2008. doi: 10.1145/1356052.1356055.

[4] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, page 90. ACM, 2013. doi: 10.1145/2503210.2503292.

[5] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 8:1–8:11. ACM, 2011. doi: 10.1145/2063384.2063394.