SLATE WORKING NOTES 9

Least Squares Performance Report

Mark Gates
Ali Charara
Jakub Kurzak
Asim YarKhan
Ichitaro Yamazaki
Jack Dongarra

Innovative Computing Laboratory

December 29, 2018

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

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 nations 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**
12-2018	first publication

```
@techreport{gates2018least,
  author={Gates, Mark and Charara, Ali and Kurzak, Jakub and
          YarKhan, Asim and Yamazaki, Ichitaro and Dongarra, Jack},
  title={{SLATE} Working Note 9: Least Squares Performance Report},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2018},
  month={December},
  number={ICL-UT-XX-XX},
  note={revision 12-2018}
}
```

Contents

Contents ii

List of Figures iii

1 Introduction 1

2 Implementation 2

2.1 Parallelization . . . 3

2.2 Least squares solver . . . 6

2.3 Deep Tile Transposition . . . 6

2.4 Handling of Multiple Precisions . . . 7

3 Experiments 10

3.1 Environment . . . 10

3.2 Performance . . . 11

4 Summary 13

Bibliography 14

List of Figures

2.1 QR factorization with lookahead of one. 3
2.2 CAQR panel factorization. 4
2.3 Local panel factorization in CAQR. 5
2.4 Trailing submatrix update in CAQR. 5
2.5 Deep conjugate-transpose 8
2.6 Deep transpose 8
2.7 Shallow conjugate-transpose 9
2.8 Shallow transpose 9

3.1 CPU performance of dgeqrf and dgels. 11
3.2 CPU performance of ScaLAPACK vs GPU performance of SLATE. 11

CHAPTER 1

Introduction

Software for Linear Algebra Targeting Exascale (SLATE) [1] [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 purpose of SLATE is to serve as a replacement for ScaLAPACK for the upcoming pre-exascale and exascale DOE machines. SLATE will accomplish this objective by leveraging recent progress in parallel programming models and by strongly focusing on supporting hardware accelerators.

This report focuses on the set of SLATE routines that solve least squares problems. Specifically, initial performance numbers are reported, alongside ScaLAPACK performance numbers, on the SummitDev machine at the Oak Ridge Leadership Computing Facility (OLCF). More details about the design of the SLATE software infrastructure can be found in the report by Kurzak et al. [1].

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

CHAPTER 2

Implementation

The principles of the SLATE software framework were laid out in SLATE Working Note 3 [1] [1]. SLATE's design relies on the following principles:

- The matrix is represented as a set of individual tiles with no constraints on their locations in memory with respect to one another. Any tile can reside anywhere in memory and have any stride. Notably, a SLATE matrix can be created from a LAPACK matrix or a ScaLAPACK matrix without making a copy of the data.
- Node-level scheduling relies on nested Open Multi Processing (OpenMP) tasking, with the top level responsible for resolving data dependencies and the bottom level responsible for deploying large numbers of independent tasks to multi-core processors and accelerator devices.
- Batch BLAS is used extensively for maximum node-level performance. Most routines spend the majority of their execution in the call to batch `gemm`.
- The Message Passing Interface (MPI) is used for message passing with emphasis on collective communication, with the majority of communication being cast as broadcasts.

Also, the use of a runtime scheduling system, such as the Parallel Runtime Scheduling and Execution Controller (PaRSEC) [2] [2] or Legion [3,4] [3], is currently under investigation.

[1] http://www.icl.utk.edu/publications/swan-003
[2] http://icl.utk.edu/parsec/
[3] http://legion.stanford.edu
[4] http://www.lanl.gov/projects/programming-models/legion.php

2.1 Parallelization

SLATE least squares solvers are marked by much higher complexity than (Sca)LAPACK due to a totally different representation of the matrix. Consider the following factors:

- SLATE matrix is a “loose” collection of tiles, i.e., there are no constraints on the memory location of any tile with respect to the other tiles. Notably, however, a ScaLAPACK matrix can still be mapped to a SLATE matrix without making a copy of the data.
- SLATE matrix can be partitioned to distributed memory nodes in any possible way, i.e., no assumptions are made about the placement of any tiles with respect to the other tiles. The same applies to the partitioning of tiles within each node to multiple accelerators.
- In principle, SLATE can support non-uniform tile sizes within the same matrix, although this mode of operation has not been well tested, as currently supporting the standard 2D block cyclic partitioning, for compatibility with ScaLAPACK, is the top priority.

Householder reflections can be used to calculate QR decompositions by reflecting first one column of a matrix onto a multiple of a standard basis vector, calculating the transformation matrix, multiplying it with the original matrix and then recursing down the (i, i) minors of that product. The standard procedure of LAPACK and ScaLAPACK is to replace each eliminated column with the coefficients of the Householder reflector. LAPACK and ScaLAPACK also apply the technique of *algorithmic blocking*, i.e., alternating steps of factoring a small set of columns (the *panel*) and applying the resulting transformations to the *trailing submatrix*.

The basic mechanics of the CAQR factorization in SLATE are shown on Figure 2.1. Like most routines in SLATE, the implementation relies on nested OpenMP tasking, where the top level is responsible for scheduling large-grained, inter-dependent tasks, and the nested level is responsible for dispatching large numbers of fine-grained, independent tasks. In the case of GPU acceleration, the nested level is implemented using calls to batch BLAS, to exploit the efficiency of processing large numbers of tiles in a call to a single GPU kernel.

Similarly to other routines, the CAQR factorization in SLATE applies the technique of *lookahead* [4–6], where one or more columns, immediately following the panel, are prioritized for faster processing, to allow for speedier advancement along the critical path. Lookahead provides large performance improvements, as it allows for overlapping the panel factorization, which is usually inefficient, with updating of the trailing submatrix, which is usually very efficient and can be GPU accelerated. Usually, the lookahead of one results in a large performance gain, while bigger values deliver diminishing returns.

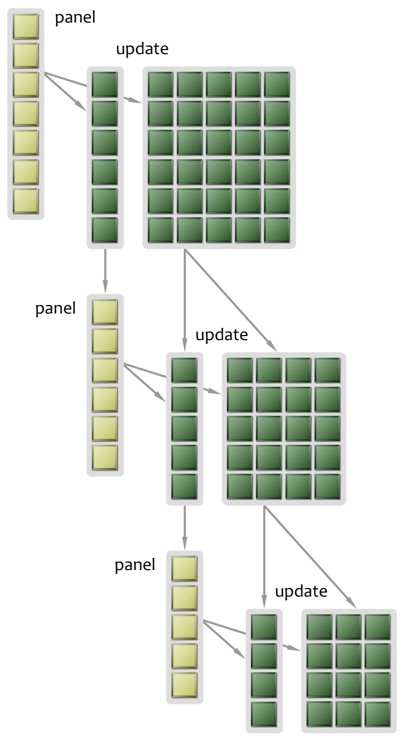


Figure 2.1: QR factorization with lookahead of one.

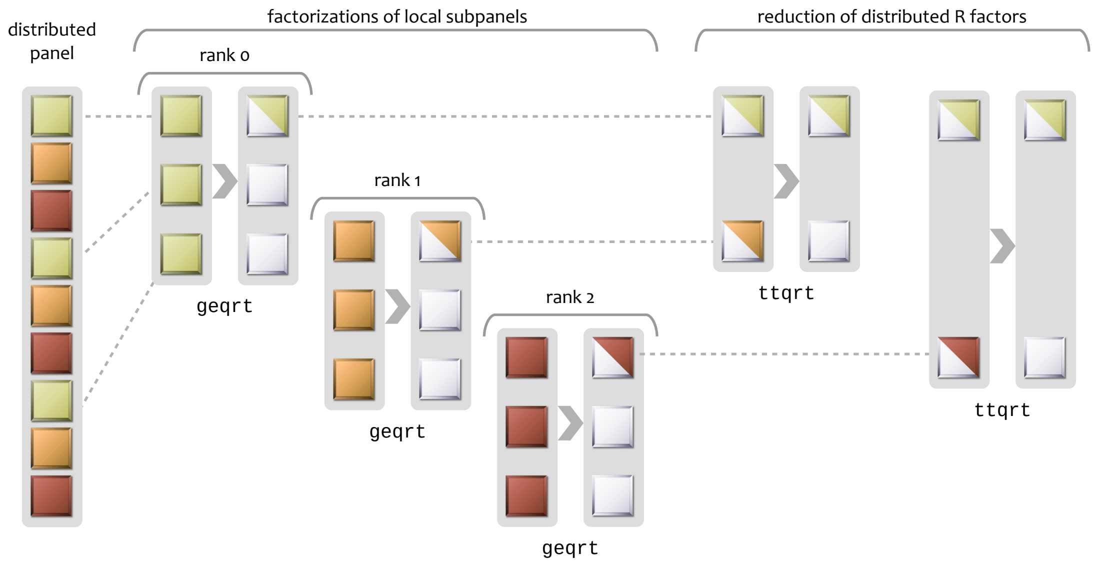

distributed panel
factorizations of local subpanels
reduction of distributed R factors
rank 0
rank 1
rank 2
geqrt
geqrt
geqrt
ttqrt
ttqrt

Figure 2.2: CAQR panel factorization.

SLATE implements the communication avoiding QR popularized by Demmel [7]. Figure 2.2 shows the basic premise of that algorithm. Here the panel is distributed in a block cyclic fashion to multiple MPI ranks. First, each rank applies the standard QR factorization to a panel consisting of its local tiles (equivalent of the LAPACK `geqrt` routine). This step requires no communication and eliminates all entries except for the upper triangular part of the top tile in each rank. The follow-up step applies a binary tree of pairwise reductions of the remaining triangles (equivalent of the LAPACK `ttqrt` routine). At the end, the upper triangular part of the topmost tile contains the R factor of the QR factorization, and the eliminated entries are replaced with coefficients of the Householder reflectors used in the elimination process. This is a different set of reflectors than the one produced by the standard QR algorithm of LAPACK and ScaLAPACK. Although, an algorithm exists for reconstructing the standard reflectors from the CAQR reflectors [8].

Within each rank, the standard QR factorization is applied to the *local panel*, i.e., the subset of tiles from the *global panel* that are mapped to that rank. The local panel factorization in SLATE relies on multithreading and internal blocking for maximum multi-core performance. Figure 2.4 shows the basic premise of the implementation. The tiles are assigned to threads in a round robin fashion, and the assignment is persistent, which allows for a high degree of cache reuse throughout the panel factorization. Also, the routine is internally blocked, i.e., the factorization of a panel of width nb proceeds in steps of much smaller width ib. While typical values of nb are 192, 256, etc., typical values of ib are 8, 16, etc. The ib factorization contains mostly BLAS 1 and 2 operations, but can benefit to some extend from cache residency, while the nb factorization contains mostly BLAS 3 operations and can also benefit from cache residency.

At each step of the ib panel factorization, a stripe of Householder reflectors is computed (V), along with a small triangular part of the R factor (R_{11}), and a small triangular part of the T factor (T_{21}). All this work is done one column at a time. What follows is application of the V reflectors to the right, which includes updating the remaining A_{22} submatrix, and computing of a new horizontal stripe of the R factor (R_{12}). Most of this work is done using BLAS 3 operations and uses the newly computed set of T factors (T_{21}). At each step, also a vertical stripe of T factors is computed (T_{11}), resulting from combining past transformations with the transformations of the current ib panel. This is also done mostly using BLAS 3 operations (`gemm` and `trmm`). This way, at the end of the nb panel factorization, a full T factor is produced, which allows for efficient application of the update to the trailing submatrix.

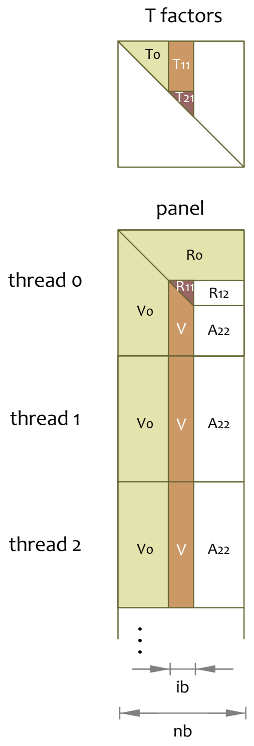


Figure 2.3: Local panel factorization in CAQR.

Updating of the trailing submatrix consists of two stages (the right side of Figure 2.4). The first one applies the transformations from the local panel factorizations and is equivalent to the LAPACK `unmqr` function. The necessary communication involves broadcast of the panel to the right. This includes the Householder reflectors and the corresponding T factors. The second stage applies a sequence of transformations resulting from the steps of the tree reduction, and is equivalent to a sequence of calls to the LAPACK `ttmqr` routine. This requires both horizontal broadcasts and point-to-point communication exchanging data between the sets of affected rows.

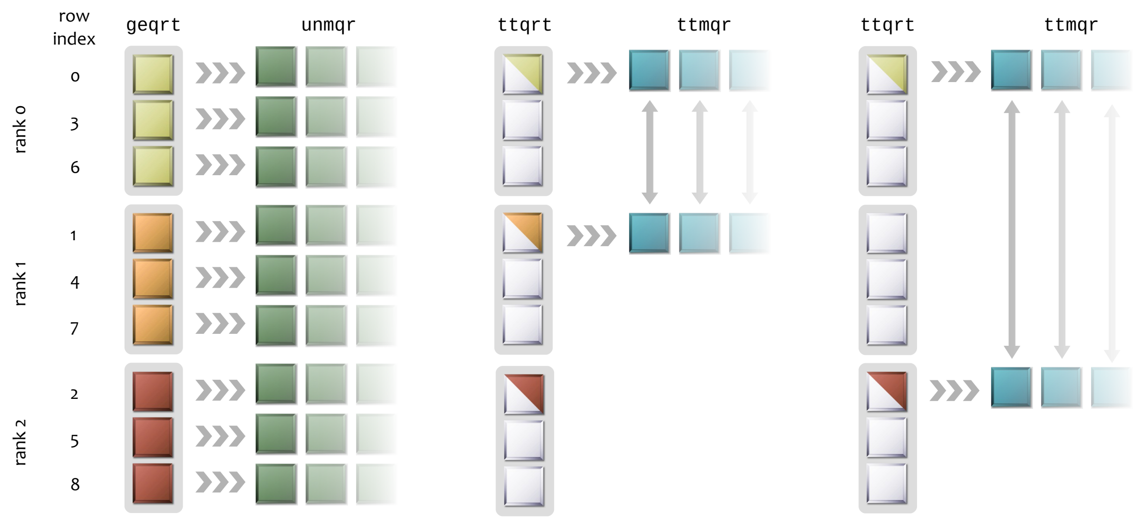


Figure 2.4: Trailing submatrix update in CAQR.

2.2 Least squares solver

The QR and LQ factorizations are used to solve the problem

$$op(A)X = B \tag{2.1}$$

where $op(A) = A$ or A^H is $m \times n$, X is $n \times nrhs$, and B is $m \times nrhs$. The various cases below are implement in SLATE in the `gels` routine.

If $m > n$, Equation (2.1) is over-determined and typically inconsistent (has no exact solution), so is solved in the least squares sense: find X that minimizes the residual,

$$\|op(A)X - B\|_2 \, . \tag{2.2}$$

For $op(A) = A$, this can be solved via a QR factorization of A, yielding $X = R^{-1}Q^H B$. In SLATE, this is implemented using `geqrf` to factor $A = QR$, `unmqr` to multiply $W = Q^H B$, then `trsm` to solve $X = R^{-1}W$. For $op(A) = A^H$, it can be solved via an LQ factorization of A, yielding $X = L^{-H}(QB)$. This case is not yet implemented in SLATE.

If $m < n$, Equation (2.1) is under-determined and typically has an infinite number of solutions, so the solution X with minimum norm is sought. For $op(A) = A^H$, this can be solved via a QR factorization, yielding $X = Q(R^{-H}B)$. In SLATE, this is implemented using `geqrf` to factor $A = QR$, `trsm` to solve $W = R^{-H}B$, then `unmqr` to multiply $X = QW$. For $op(A) = A$, it can be solved via an LQ factorization of A, yielding $X = Q^H(L^{-1}B)$. This case is not yet implemented in SLATE.

If A is rank deficient, the above QR technique will fail because the triangular matrix R will be singular. In that case, other techniques such as rank-revealing QR or the singular value decomposition (SVD) are applicable. These techniques will be addressed by future SLATE developments.

2.3 Deep Tile Transposition

As evident from Section 2.2, the over- and under-determined problems can both be solved by either QR or LQ. In fact, QR and LQ are simply conjugate-transposes of each other:

$$(QR)^H = R^H Q^H = L\hat{Q}.$$

The QR factorization forms Householder reflectors to eliminate each column below the diagonal, hence accesses data column-wise, while the LQ factorization applies Householder reflectors to eliminate each row right of the diagonal, hence accesses data row-wise. Since SLATE by default stores data in column-major order, accessing data row-wise in LQ would be inefficient. Also, writing an LQ routine that is basically identical to the QR routine but applied row-wise would introduce undesired code duplication.

Instead, to compute an LQ factorization we employ transposition and compute the QR factorization of A^H, then transpose the resulting QR back to obtain $L\hat{Q}$. For most purposes, SLATE uses a shallow transposition, which merely marks a matrix and its tiles as transposed, without physically transposing data in memory. The underlying BLAS routines (`gemm`, etc.) take the transposition flag and apply it during the computation. However, in LQ, this shallow transpose would still leave inefficient row-wise access to column-major data. Instead, we employ a deep transpose that physically transposes the tiles in memory.

Each tile is transposed independently. Square tiles can always be transposed in place. Rectangular tiles, which occur on the border of the matrix, must be contiguous, not strided, to be transposed in place. If data starts in ScaLAPACK format, we handle making just the border tiles contiguous in the Matrix `fromScaLAPACK` constructor, and copying the border tiles back to ScaLAPACK format via `toScaLAPACK`. As with the shallow transpose, accessing tiles swaps indices, so accessing tile $A^H(i, j)$ returns tile $A(j, i)$.

When shallow and deep transpose are combined, it leads to several mixed states, such as $(A^T)^t$, where capital T represents deep transpose and t represents shallow transpose. Mixing a shallow transpose and shallow conjugate-transpose is prohibited, since BLAS does not support it, but otherwise all combinations are allowed. The complete state transitions are shown in Figures 2.5 to 2.8. There are multiple representations of the same logical matrix:

no transpose: A, $(A^T)^t$, and $(A^H)^h$ (light grey states)

conjugate: A^*, $(A^H)^t$, and $(A^T)^h$ (dark grey states)

transpose: A^t, A^T, and A^{*h} (blue states)

conjugate-transpose: A^h, A^H, and A^{*t} (orange states)

2.4 Handling of Multiple Precisions

SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. SLATE's LAPACK++ component [9] provides overloaded, precision-independent wrappers for all the underlying LAPACK routines, which SLATE's least squares solvers are built on top of. For instance, `lapack::tpqrt` in LAPACK++ maps to `stpqrt`, `dtpqrt`, `ctpqrt`, or `ztpqrt` LAPACK routines, depending on the precision of its arguments.

Where a data type is always real, `blas::real_type<scalar_t>` is a C++ type trait to provide the real type associated with the type `scalar_t`, so `blas::real_type< std::complex<double> >` is `double`. Since norms of complex matrices are real values, this is used across the norms routines.

Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE norms code should be able to accommodate other data types, such as quad precision, given appropriate implementations of the elemental operations.

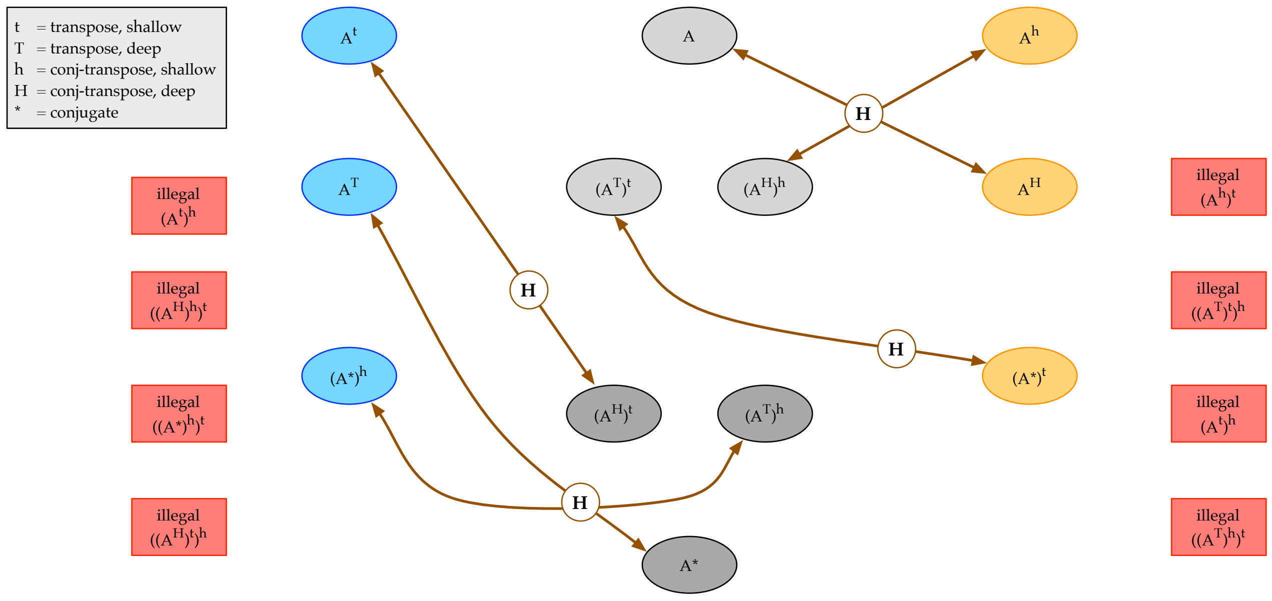


Figure 2.5: Deep conjugate-transpose

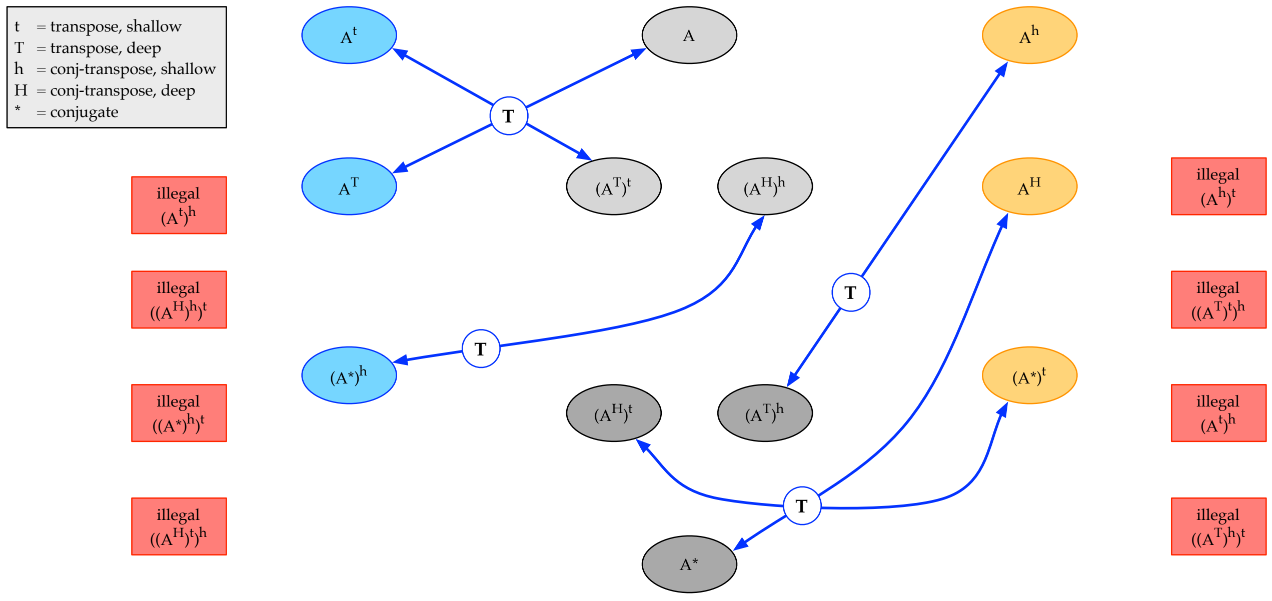


Figure 2.6: Deep transpose

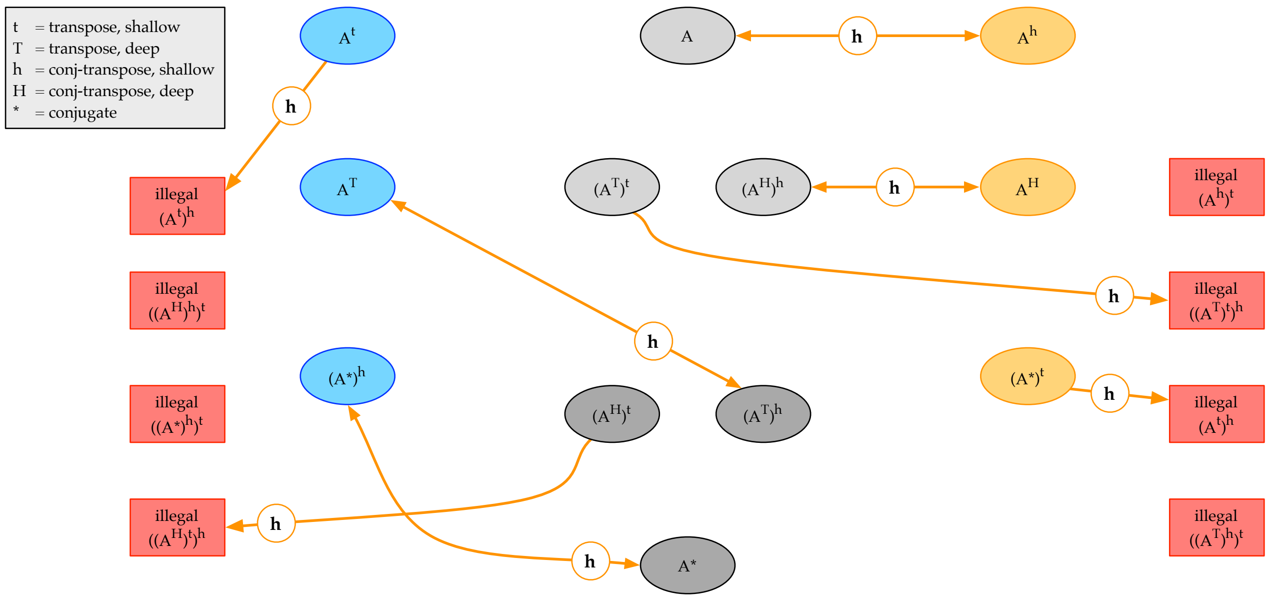


Figure 2.7: Shallow conjugate-transpose

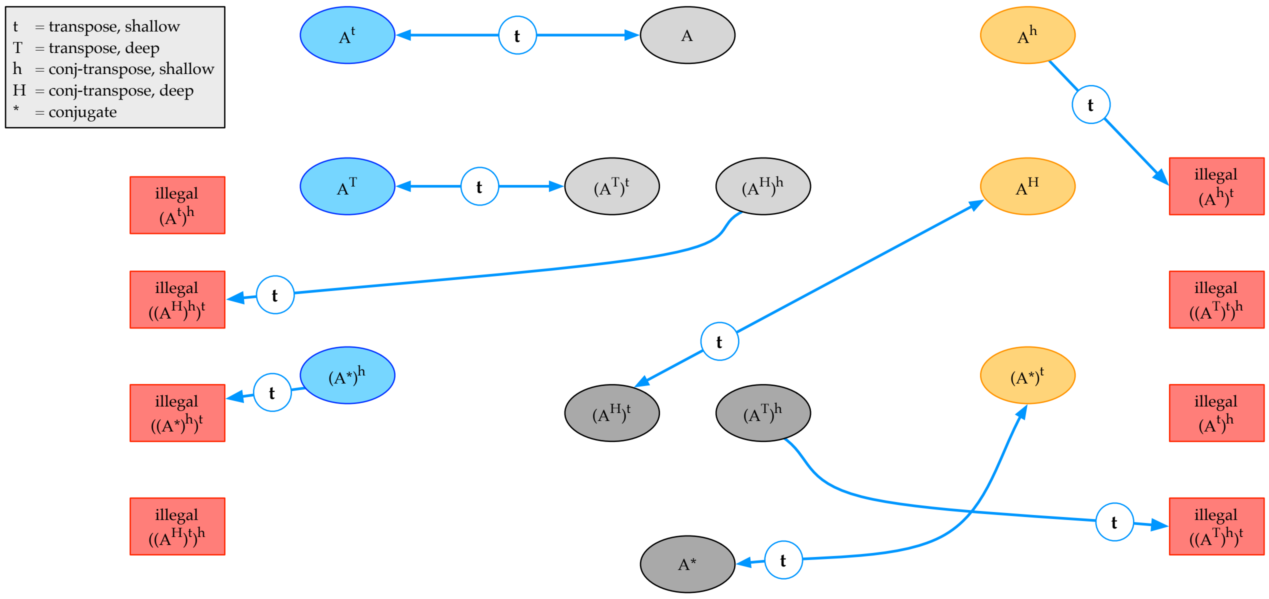


Figure 2.8: Shallow transpose

CHAPTER 3

Experiments

3.1 Environment

Performance numbers were collected using the SummitDev system [1] at the OLCF, which is intended to mimic the OLCF's next supercomputer, Summit. SummitDev is based on IBM POWER8 processors and NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which will be based on IBM POWER9 processors and NVIDIA V100 (Volta) accelerators.

The SummitDev system contains three racks, each with eighteen IBM POWER8 S822LC nodes, for a total of fifty-four nodes. Each node contains two POWER8 CPUs, ten cores each, and four P100 GPUs. Each node has 256 GB of DDR4 memory. Each GPU has 16 GB of HBM2 memory. The GPUs are connected by NVLink 1.0 at 80 GB/s. The nodes are connected with a fat-tree enhanced data rate (EDR) InfiniBand.

The software environment used for the experiments included GNU Compiler Collection (GCC) 7.1.0, CUDA 9.0.69, Engineering Scientific Subroutine Library (ESSL) 5.5.0, Spectrum MPI 10.1.0.4, Netlib LAPACK 3.6.1, and Netlib ScaLAPACK 2.0.2—i.e., the output of `module list` included:

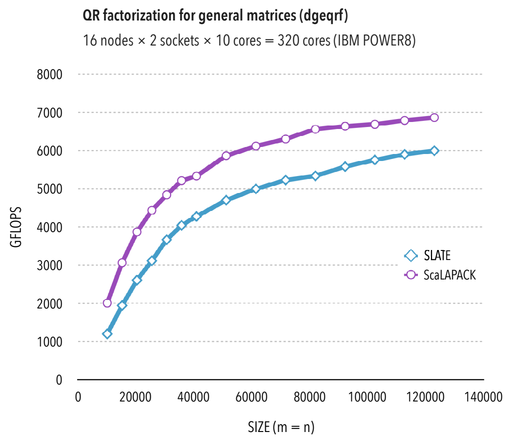
```
gcc/7.1.0
cuda/9.0.69
essl/5.5.0-20161110
spectrum-mpi/10.1.0.4-20170915
netlib-lapack/3.6.1
netlib-scalapack/2.0.2
```

[1]https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/

3.2 Performance

All runs were performed using sixteen nodes of the SummitDev system, which provides $16\ nodes \times 2\ sockets \times 10\ cores = 320$ IBM POWER8 cores and $16\ nodes \times 4\ devices = 64$ NVIDIA P100 accelerators. SLATE was run with one process per node, while ScaLAPACK was run with one process per core, which is still the prevailing method of getting the best performance from ScaLAPACK. Only rudimentary performance tuning was done in both cases.

Figure 3.1 shows the CPU performance of ScaLAPACK and SLATE for the `dgeqrf` routine (QR factorization) and the `dgels` routine (least squares solve). The horizontal axis shows the problem size with $m = n$ for the QR factorization and $m = n = nrhs$ for the least squares solve. Figure 3.2 shows the comparison of ScaLAPACK's CPU performance to SLATE's GPU performance.



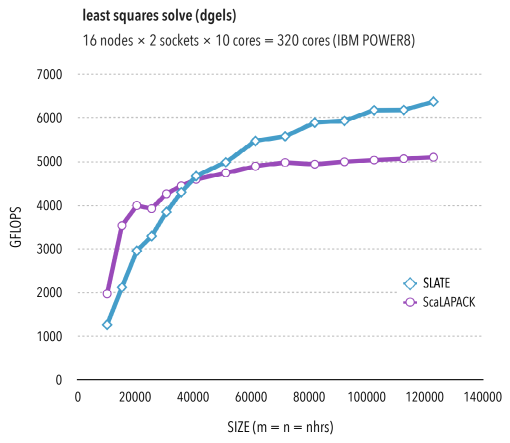
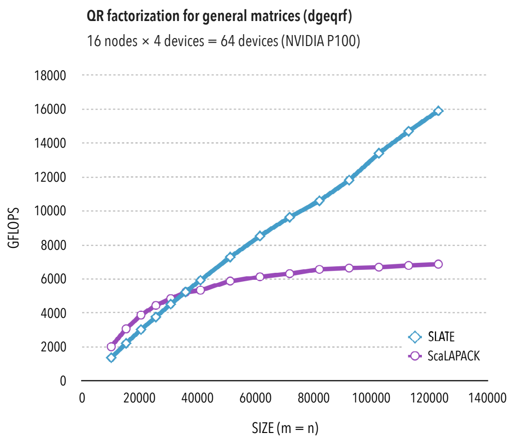


Figure 3.1: CPU performance of `dgeqrf` (left) and `dgels` (right).



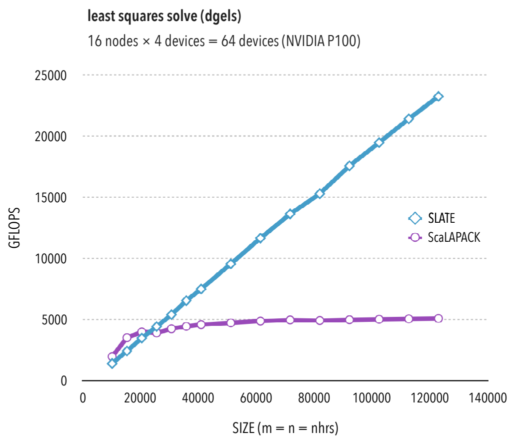


Figure 3.2: CPU performance of ScaLAPACK vs GPU performance of SLATE.

The CPU performance of SLATE's `dgeqrf` is slightly lower than ScaLAPACK's, while the CPU performance of SLATE's `dgels` is slightly higher than ScaLAPACK's. Overall, for CPU runs, SLATE delivers very similar performance to ScaLAPACK.

GPU performance of SLATE is superior to CPU performance of ScaLAPACK, with up to $2.5\times$ speedup for `dgeqrf` and up to $5\times$ speedup for `dgels`. This, however, is nowhere near the expected order of magnitude speedup and clearly more performance engineering is needed.

CHAPTER 4

Summary

Implementation of least squares solvers in SLATE faced multiple challenges. Development of multithreaded panel factorization, implementation of tree reductions, and introduction of deep transposition of tiles were all non-trivial tasks.

SLATE's software infrastructure provided the flexibility to accommodate the necessary changes, and the resulting routines provide competitive CPU performance and moderate GPU acceleration. More performance engineering is needed for maximum GPU utilization.

Bibliography

[1] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. SLATE working note 3: Designing SLATE: Software for linear algebra targeting exascale. Technical Report ICL-UT-17-06, Innovative Computing Laboratory, University of Tennessee, September 2017. revision 09-2017.

[2] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[4] Peter Strazdins et al. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. 1998.

[5] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *International Workshop on Applied Parallel Computing*, pages 147–156. Springer, 2006.

[6] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[7] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-avoiding parallel and sequential QR and LU factorizations. In *SIAM Journal of Scientific Computing*. Citeseer, 2008.

[8] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and

Edgar Solomonik. Reconstructing Householder vectors from tall-skinny QR. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1159–1170. IEEE, 2014.

[9] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. SLATE working note 2: C++ API for BLAS and LAPACK. Technical Report ICL-UT-17-03, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 03-2018.