Least Squares Solvers for Distributed-Memory Machines with GPU Accelerators

Jakub Kurzak
University of Tennessee
Knoxville, Tennessee
kurzak@icl.utk.edu

Mark Gates
University of Tennessee
Knoxville, Tennessee
mgates3@icl.utk.edu

Ali Charara
University of Tennessee
Knoxville, Tennessee
charara@icl.utk.edu

Asim YarKhan
University of Tennessee
Knoxville, Tennessee
yarkhan@icl.utk.edu

Jack Dongarra*
University of Tennessee
Knoxville, Tennessee
dongarra@icl.utk.edu

ABSTRACT

This work presents an implementation of a linear least squares solver for distributed-memory machines with GPU accelerators, developed as part of the Software for Linear Algebra Targeting Exascale (SLATE) package. From the algorithmic standpoint, the work leverages recent advances in dense linear algebra, specifically the communication-avoiding QR factorization. From the implementation standpoint, the work represents a sharp departure from the traditional conventions established by legacy packages, such as LAPACK and ScaLAPACK. It is based on representing the matrix as a collection of individual tiles, and using batch operations for offloading work to accelerators. The article lays out the principles of the new approach, discusses the implementation details and presents the performance results.

CCS CONCEPTS

• **Mathematics of computing** → **Computations on matrices**; • **Computing methodologies** → **Linear algebra algorithms**; **Massively parallel algorithms**.

KEYWORDS

linear algebra, distributed memory, least squares

ACM Reference Format:
Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. Least Squares Solvers for Distributed-Memory Machines with GPU Accelerators. In *Proceedings of International Conference on Supercomputing (ICS'19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3330345.3330356

*Jack Dongarra also holds appointments at Oak Ridge National Laboratory and the University of Manchester.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330356

1 MOTIVATION

There is an urgent need for multi-GPU–accelerated, distributed-memory software. In the United States, the plan for achieving the Exascale relies heavily on the use of GPU-accelerated machines, similar to the Summit[1] and Sierra[2] systems at Oak Ridge National Laboratory (ORNL) and Lawrence Livermore National Laboratory (LLNL), respectively, which currently occupy positions #1 and #2 on the TOP500 list. Although an alternative path exists—the Intel A21 system, Aurora, planned for the Argonne National Laboratory (ANL)[3] [4]—no public information about the architecture has been released so far.

The urgency of the situation is underscored by the architectures of the aforementioned systems.[5] The Summit system contains three NVIDIA V100 GPUs per each POWER9 CPU. The peak double-precision floating-point performance of the CPU is 22 (cores) × 24.56 GFLOPS = 540.32 GFLOPS. The peak performance of the GPUs is 3 (devices) × 7.8 TFLOPS = 23.4 TFLOPS, i.e., 97.7% of performance is on the GPU side, and only 2.3% of performance is on the CPU side.

Considering the huge gap between the computing power of systems like Summit and Sierra and their interconnection technology, there is a strong motivation for pursuing algorithmic innovations that aim to minimize communication [5, 12]. At the same time, the disparity between the capabilities of the CPUs and the capabilities of the GPUs provides a strong incentive to aggressively optimize the CPU tasks residing in the critical path of the algorithm, using solutions for efficient multithreading and cache efficiency [10].

Here we present the only implementation, that we know of, that targets Summit- and Sierra-class machines, i.e., large distributed-memory systems drawing virtually all of their computing power from GPU accelerators. Our implementation is based on the infrastructure of the Software for Linear Algebra Targeting Exascale (SLATE) project, which is a radical departure from the established

[1] https://www.olcf.ornl.gov/summit/
[2] https://hpc.llnl.gov/hardware/platforms/sierra
[3] https://www.nextplatform.com/2017/11/14/looking-ahead-intels-secret-exascale-architecture/
[4] https://www.nextplatform.com/2018/03/19/argonne-hints-at-future-architecture-of-aurora-exascale-system/
[5] https://en.wikichip.org/wiki/supercomputers/summit

conventions, most notably from the legacy matrix layout of ScaLAPACK. Moreover, as far as we know, we produced a unique implementation of the QR panel factorization, which combines internal blocking, cache residency, and multithreading.

2 BACKGROUND

2.1 Least Squares

The linear least squares solver is one of the main tools of linear regression, which is widely used in many disciplines of science and technology. It ranks as one of the most important tools in behavioral and biological sciences—bioinformatics, finance, economics, environmental science, etc. Linear regression also plays an important role in artificial intelligence, where the linear regression algorithm is one of the fundamental supervised machine learning algorithms due to its relative simplicity and well-known properties.

The linear least squares problem is the problem of finding an approximate solution to an overdetermined system of equations $Ax = b$, a system where the number of equations is larger than the number of unknowns, such that $\|b - Ax\|^2$ is minimized. It is most commonly solved using QR factorization, which is a decomposition of a matrix A into a product $A = QR$ of an orthogonal matrix Q and an upper triangular matrix R. There are several methods for computing the QR decomposition, such as by means of the Gram-Schmidt process, Householder transformations, or Givens rotations, each one with a number of advantages and disadvantages. This work relies on the use of Householder reflections, which is the solution adopted by the LAPACK [3] and ScaLAPACK [7] software packages.

A related problem is the one of solving an underdetermined system of equations, i.e., a system where the number of equations is smaller than the number of unknowns. While, in this case, many solutions exist, we can pick the one with the smallest $\|x\|^2$. This problem can be solved by using the LQ factorization, which is a decomposition of a matrix A into a product $A = LQ$ of a lower triangular matrix L and an orthogonal matrix Q. The Householder procedure for finding the LQ decomposition is basically identical to the QR procedure.

2.2 SLATE Project

SLATE[6] is being developed as part of the Exascale Computing Project (ECP),[7] which is a collaborative effort between two US Department of Energy (DOE) organizations, the Office of Science and the National Nuclear Security Administration (NNSA). The objective of SLATE is to provide fundamental dense linear algebra capabilities to the US Department of Energy and to the high-performance computing (HPC) community at large.

The ultimate objective of SLATE is to replace the ScaLAPACK library, which has become the industry standard for dense linear algebra operations in distributed-memory environments. However, after two decades of operation, ScaLAPACK is past the end of its life cycle and overdue for a replacement, as it can hardly be retrofitted to support hardware accelerators, which are an integral part of today's HPC hardware infrastructure.

[6] http://icl.utk.edu/slate/
[7] https://www.exascaleproject.org

For context, ScaLAPACK was first released in 1995, some 24 years ago. In the past two decades, HPC has witnessed tectonic shifts in the hardware technology, followed by paradigm shifts in the software technology, and a plethora of algorithmic innovations in scientific computing. At the same time, no viable replacement for ScaLAPACK emerged. SLATE is meant to be this replacement, boasting superior performance and scalability in the modern, heterogeneous, distributed-memory environments of HPC.

Primarily, SLATE aims to extract the full performance potential and maximum scalability from modern, many-node HPC machines with large numbers of cores and multiple hardware accelerators per node. For typical dense linear algebra workloads, this means getting close to the theoretical peak performance and scaling to the full size of the machine (i.e., thousands to tens of thousands of nodes). This is to be accomplished in a portable manner by relying on standards like MPI and OpenMP.

SLATE functionalities will first be delivered to the ECP applications that most urgently require SLATE capabilities (e.g., EXascale Atomistics with Accuracy, Length, and Time [EXAALT], NorthWest computational Chemistry for Exascale [NWChemEx], Quantum Monte Carlo PACKage [QMCPACK], General Atomic and Molecular Electronic Structure System [GAMESS], CANcer Distributed Learning Environment [CANDLE]) and to other software libraries that rely on underlying dense linear algebra services (e.g., Factorization Based Sparse Solvers and Preconditioners [FBSS]). SLATE will not only fill the void left by ScaLAPACK's inability to utilize hardware accelerators, but will also ease the difficulties associated with ScaLAPACK's legacy matrix layout and Fortran API. Figure 1 shows SLATE in the ECP software stack.

3 RELATED WORK

Over the past two decades, optimization of QR factorization received a lot of attention. From the standpoint of this article, the most important directions included: efforts to improve the serial performance through better cache utilization [6, 15, 16, 24], work on minimizing communication in distributed-memory environments [5, 12], research on efficient scheduling and critical path minimization [8, 13], and solutions to the problem of multithreading of the memory-bound operations [10].

Fast serial implementation of the QR factorization is the foundation for all other optimizations. Seminal work in this area was done by Bischof, Schreiber, and Van Loan, who introduced the *WY* representation for products of Householder reflectors [6, 24], an idea which became the cornerstone of the QR routines in LAPACK and ScaLAPACK [11]. A closely related development is the introduction of recursive QR factorization by Elmroth and Gustavson [15, 16].

The seminal work on avoiding communication in the QR factorization was done by Demmel et al. [12], and later generalized to other dense linear algebra algorithms. The article by Ballard et al. [5] is a compendium of knowledge on the topic. The implementation presented in this article is primarily based on Demmel's original idea of splitting the panel factorization into multiple pieces and following up with a tree reduction. A related development is the study of different reduction patterns and their corresponding critical paths. Notable contributions include the work by Bouwmeester et al. [8] and Dongarra et al. [13].

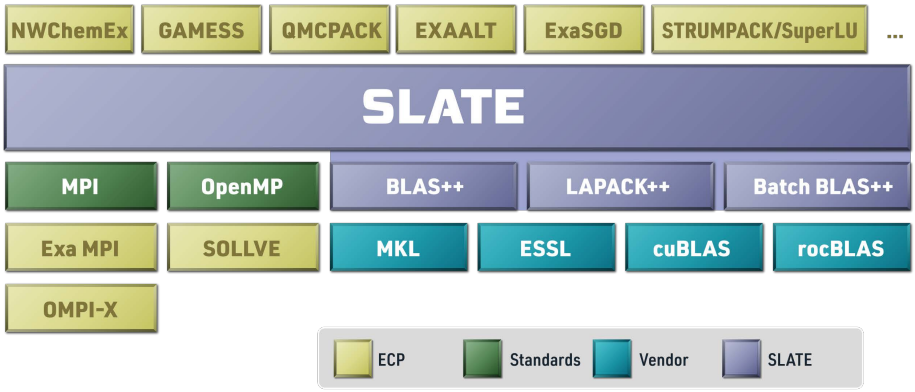

Figure 1: SLATE in the ECP software stack.

A significant amount of effort has been put towards addressing the inefficiencies of the operation know as the *panel factorization*, i.e., factorization of a single stripe of the input matrix in order to accumulate multiple transformations and apply them using efficient, Level 3 BLAS operations to the remaining submatrix. Castaldo et al. [10] proposed the technique of *parallel cache assignment* for preserving cache efficiency while applying multithreading. Dongarra et al. [14] used a similar technique, while also applying recursion, to the panel factorization in the Gaussian elimination.

Other relevant developments include work on the tile QR factorization [8, 9], which reduces the matrix tile by tile and leads to very efficient, highly pipelined multi-core implementations. However, while relevant, the tile QR is not directly related to this work. Though it is attractive from the standpoint of strong scaling, the algorithm has not been chosen for SLATE, as its asymptotic performance is handicapped by complex kernels, i.e., it cannot easily leverage the performance of the batched `gemm` kernel [23].

4 IMPLEMENTATION

4.1 Matrix Storage

The matrix storage in SLATE is a radical departure from the conventions of LAPACK and ScaLAPACK. Unlike the legacy packages, which store the matrix contiguously, by columns, SLATE stores the matrix as a collection of individual tiles. This offers numerous advantages, e.g.,:

- The same structure can be used for holding many different matrix types,[8] e.g., general, symmetric, triangular, band, symmetric band, etc. No memory is wasted for storing parts of the matrix that hold no useful data, e.g., the upper triangle of a lower triangular matrix. There is no need for using complex matrix layouts, such as the Recursive Packed Format (RPF) [1, 2, 18] in order to save space.
- The matrix can be easily converted, in parallel, from one layout to another with $O(P)$ memory overhead, where P is the number of processors (cores/threads) used. Possible conversions include: changing the layout of tiles from column major to row major, "packing" of tiles for efficient execution of the `gemm` operation,[9] low-rank compression of tiles, re-tiling of the matrix (changing the tile size), etc. Notably, transposition of the matrix can be accomplished by transposition of each tile and remapping of the indices. There is no need for complex in-place layout translation and transposition algorithms [19].
- Tiles can easily be moved or copied among different memory spaces. Both inter-node communication and intra-node communication is vastly simplified. Tiles can easily and efficiently be transferred between nodes using MPI. Tiles can be easily moved in and out of faster memory, such as the MCDRAM in the Xeon Phi processors. Tiles can also be copied to one or more device memories in the case of GPU acceleration.

In practical terms, the SLATE matrix is implemented by the `std::map` container from the standard C++ library, i.e.:

```
std::map< std::tuple< int64_t, int64_t, int >,
          Tile<scalar_t>* >
```

The key is a triplet consisting of the (i, j) position of the tile in the matrix and the device number where the tile is located. The value is a pointer to an object of a lightweight class that stores the tile's data and its properties.

In addition to facilitating the storage of different types of matrices, this structure also readily accommodates partitioning of the matrix to the nodes of a distributed-memory system. Tile indexing is global, and each node stores only its local subset of tiles. Mapping of tiles to nodes is defined by a C++ lambda function, and set to 2D block cyclic mapping by default. Remote access is realized by mirroring remote tiles in the local matrix for the duration of the operation. In that respect, SLATE follows the single program, multiple data (SPMD) programming style and mimics the partitioned global address space (PGAS) programming model. SLATE also has the potential to support matrices with non-uniform tile sizes in the future.

[8] http://www.netlib.org/lapack/lug/node24.html

[9] https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm

4.2 Matrix Class Hierarchy

SLATE has the matrix classes below. The SLATE routines require the correct matrix types for their arguments, but inexpensive shallow copy conversions exist between the various matrix types. For instance, a general `Matrix` can be converted to a `TriangularMatrix` for doing a triangular solve (`trsm`).

BaseMatrix Abstract base class for all matrices.
- **Matrix** General, $m \times n$ matrix.
- **BaseTrapezoidMatrix** Abstract base class for all upper or lower trapezoid storage, $m \times n$ matrices. For upper, tiles $A(i, j)$ for $i \leq j$ are stored; for lower, tiles $A(i, j)$ for $i \geq j$ are stored.
  - **TrapezoidMatrix** Upper or lower trapezoid, $m \times n$ matrix; the opposite triangle is implicitly zero.
    - **TriangularMatrix** Upper or lower triangular, $n \times n$ matrix.
  - **SymmetricMatrix** Symmetric, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is implicitly known by symmetry ($A_{j,i} = A_{i,j}$).
  - **HermitianMatrix** Hermitian, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is implicitly known by symmetry ($A_{j,i} = \bar{A}_{i,j}$).

The `BaseMatrix` class stores the matrix dimensions; whether the matrix is upper, lower, or general; whether it is not transposed, transposed, or conjugate-transposed; how the matrix is distributed; and the set of tiles.

Copying a matrix object is an inexpensive shallow copy, with a reference-counted C++ shared pointer to the actual data. Submatrices are also implemented by creating an inexpensive shallow copy, with the matrix object storing the offset from the top-left of the original matrix and the transposition operation with respect to the original matrix.

4.3 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 [17] provides overloaded, precision-independent wrappers for all the underlying LAPACK routines, which SLATE's least squares routines 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`.

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

4.4 Message Passing Communication

Communication in SLATE relies on explicit dataflow information. When a tile is needed for computation, it is broadcast to all the processes where it is required. Rather than explicitly listing MPI ranks, the broadcast is expressed in terms of the destination tiles to be updated. The communication function takes a tile's (i, j) indices and a sub-matrix that the tile will update; the tile is sent to all processes owning that sub-matrix. To optimize communication, a list of communications is created and the MPI is pipelined with CPU-to-accelerator transfers. As the set of processes involved is dynamically determined from the sub-matrix, using an MPI broadcast would require setting up a new MPI communicator, which is an expensive global blocking operation. Instead, SLATE uses point-to-point MPI communication in a hypercube fashion to broadcast the data.

4.5 Node-Level Memory Consistency

Several solutions are available for dealing with the complexity of node-level memory architecture involving separate physical memories of multiple hardware accelerators. We investigated CUDA managed memory, OpenMP directives, and the OpenMP offload API. None of these seemed yet fully capable and portable across a wide variety of platforms.

Instead, SLATE allocates and manages accelerator memory itself. To support multiple accelerator devices, SLATE allows for multiple copies of a tile in different device memories. The initial copy of a local tile given by the user is marked as *origin*. This can be either in host memory or accelerator memory. All other copies are marked as workspace—either a temporary copy of a remote tile, or a copy of a local tile on another device. By default, at the end of a computation SLATE ensures that the origin copy of a tile is up-to-date, and that workspace tiles have been deleted.

For offload to GPU accelerators, SLATE implements a memory consistency model, inspired by the MOSI cache coherency protocol [20, 26], on a tile-by-tile basis. For read only access, tiles are mirrored in the memories of possibly multiple GPU devices, and deleted when no longer needed. For write access, tiles are migrated to the GPU memory and returned to the CPU memory afterwards if needed. SLATEâĂŹs memory consistency model has three states plus an orthogonal OnHold flag:

Modified (M) Tile's data is modified; other instances are Invalid; tile cannot be purged.
Shared (S) Tile's data is up-to-date with other instances; other instances may be Shared or Invalid; instance may be purged unless OnHold.
Invalid (I) Tile's data is invalid.
OnHold (O) Flag to prevent tile from being purged.

The states are managed by a simple API to fetch tiles to the CPU or accelerator as needed. The OnHold flag is used to optimize CPU to accelerator communication. Normally, at the end of an operation, workspace tiles are deleted from GPU devices to limit the required workspace memory. However, if we know that a tile will soon be used again by another operation, placing it on hold will prevent it from being purged, eliminating the subsequent re-fetching of data to the accelerator. This happens, for instance, in applying a block Householder reflector, $I - VTV^H$, where the tiles in V are used in two `gemm` operations.

4.6 QR Factorization

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 communication-avoiding QR (CAQR) factorization in SLATE are shown on Figure 2. Like most routines in SLATE, the implementation relies on nested OpenMP tasking, where the top level is responsible for scheduling large-grained, interdependent 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 batched 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* [21, 22, 25], 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.

SLATE implements the communication-avoiding QR popularized by Demmel [12]. Figure 3 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 [4].

Within each rank, the standard QR factorization is applied to the *local panel*, i.e., the subset of tiles from the *global panel*, which are mapped to that rank. The local panel factorization in SLATE relies on multithreading and internal blocking for maximum multi-core performance. Figure 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 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

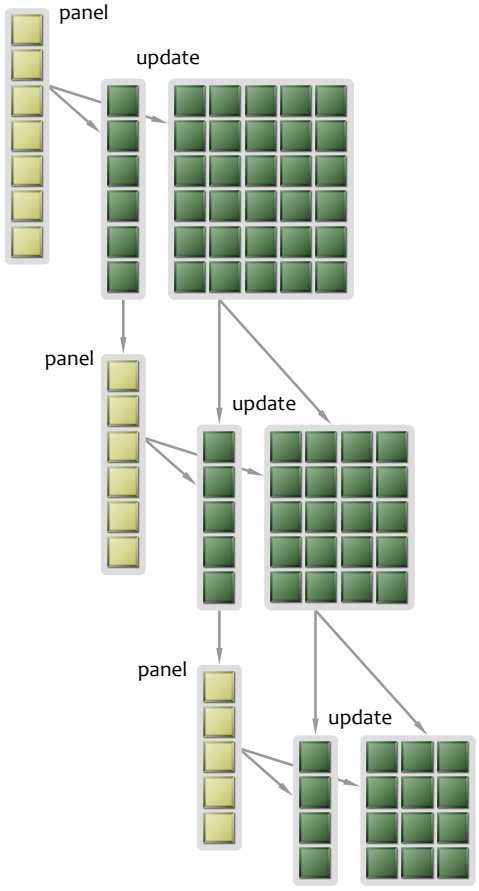

Figure 2: Basic mechanics of QR factorization in SLATE with nested parallelism and lookahead.

of nb are 192, 256, etc., typical values of ib are 8, 16, etc. The ib factorization contains mostly Level 1 and 2 BLAS operations, but can benefit to some extent 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 a new horizontal stripe of the R factor (R_{12}). Most of this work is done using Level 3 BLAS operations and uses the newly computed set of T factors (T_{21}). At each step, a vertical stripe of T factors is also computed (T_{11}), resulting from combining past transformations with the transformations of the current ib panel. This is also done mostly using Level 3 BLAS 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.

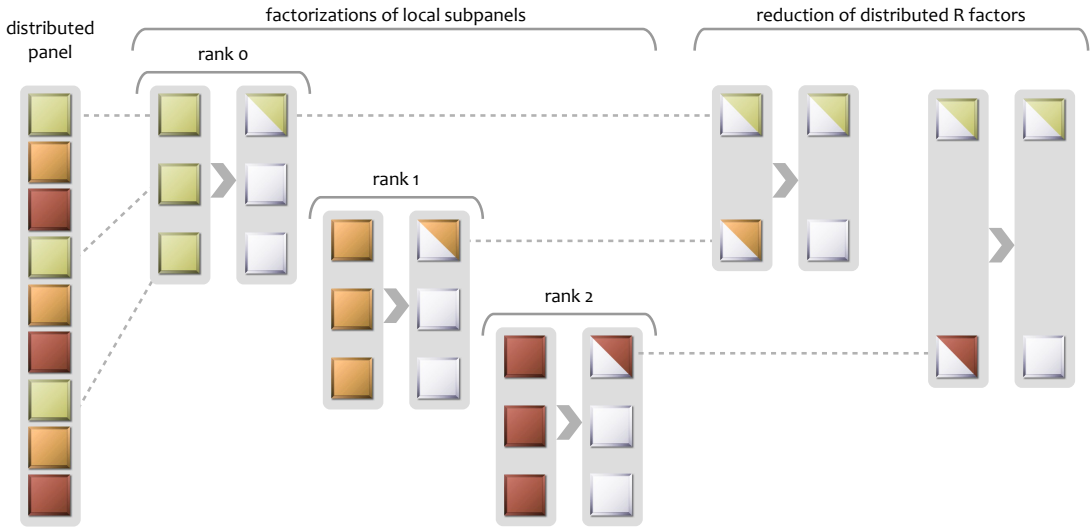

Figure 3: Stages of the panel factorization in the CAQR algorithm.

Updating of the trailing submatrix consists of two stages (Figure 5). The first 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.

4.7 Least Squares Solver

The QR and LQ factorizations are used to solve the problem

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

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 implemented in SLATE in the `gels` routine.

If $m > n$, eq. (1) is over-determined and typically inconsistent (has no exact solution), so it is solved in the least squares sense: find X that minimizes the residual,

$$||op(A)X - B||_2. \tag{2}$$

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 $m < n$, eq. (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.

4.8 Deep Tile Transposition

As evident from Section 4.7, 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, thus accessing data column-wise, while the LQ factorization applies Householder reflectors to eliminate each row right of the diagonal, thus accessing 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

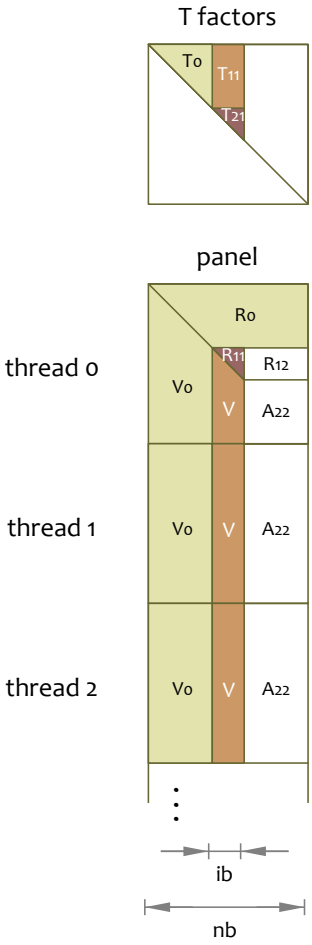

Figure 4: Local QR panel factorization in SLATE with multi-threading and internal blocking.

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.

4.9 ScaLAPACK Compatibility

Because many applications have significant existing code bases using ScaLAPACK, we provide a ScaLAPACK compatibility API. These routines intercept calls to ScaLAPACK and redirect them to SLATE calls. This is enabled simply by adjusting the link line of the application, putting `-lslate_scalapack_api` before `-lscalapack`. No code modifications are required. Any ScaLAPACK calls that SLATE implements will be intercepted; other calls will continue to use ScaLAPACK as before.

It should be noted, however, that an optimal configuration for ScaLAPACK—typically one MPI process per core—is not optimal for SLATE, where we've found using one MPI process per socket works better. Also, SLATE typically uses larger tile sizes, particularly when using accelerators. Instead of $nb = 64$ used for ScaLAPACK, SLATE may require $nb = 192$ or 256 for good efficiency.

5 PERFORMANCE ANALYSIS

5.1 Setup

Performance numbers were collected using the SummitDev system[10] at the OLCF, which is intended to mimic the OLCF's much larger supercomputer, Summit. SummitDev is based on the IBM POWER8 processors and the NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which is based on the IBM POWER9 processors and the 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.

5.2 Performance

All runs were performed using sixteen nodes of the SummitDev system, which provides 16 nodes × 2 sockets × 10 cores = 320 IBM POWER8 cores and 16 nodes × 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.

Figures 6 and 7 show performance comparisons of SLATE and ScaLAPACK using CPUs only. Figure 6 shows performance of the QR factorization (`dgeqrt`) and Figure 7 shows performance of the least square solver (`dgels`). All results are in double precision.

Figures 8 and 9 show performance comparisons of SLATE with GPU acceleration and ScaLAPACK without GPU acceleration. We are not aware of an effective way of GPU accelerating ScaLAPACK. Figure 8 shows performance of the QR factorization (`dgeqrt`) and Figure 9 shows performance of the least square solver (`dgels`). The results are also in double precision here.

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

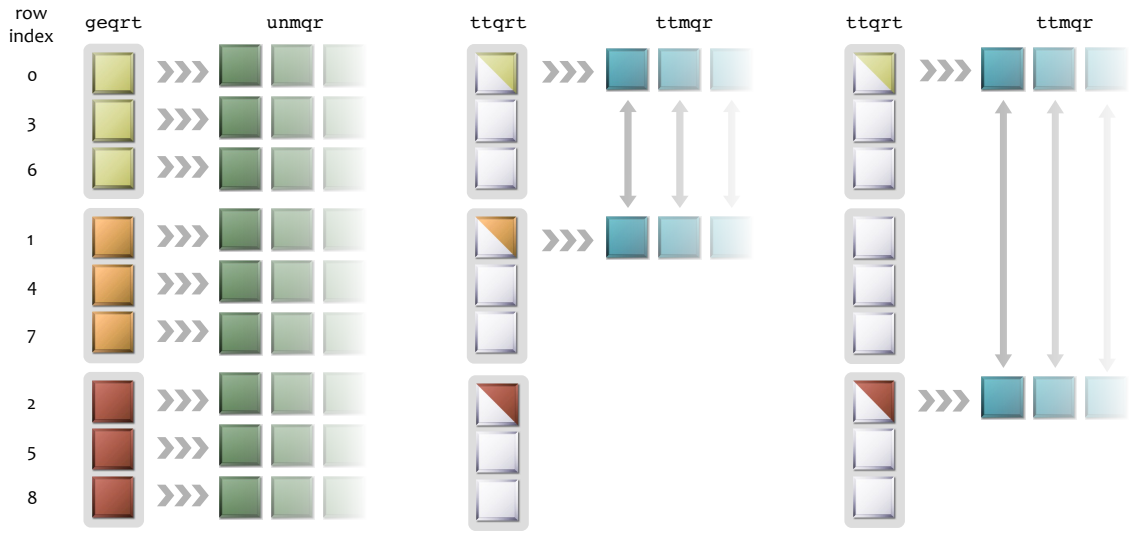

Figure 5: Diagram of the CAQR factorization in SLATE showing steps of the trailing submatrix update, including the corresponding communication.

At this point, the CPU performance of SLATE is comparable to ScaLAPACK—slightly lower for the QR factorization alone, slightly faster for the complete least squares solver. This is an initial implementation and numerous performance improvement opportunities exist: faster kernels, improved scheduling, improved communication. We expect SLATE to eventually surpass ScaLAPACK across the entire spectrum.

At the same time, SLATE clearly shows the capability of benefiting from GPU acceleration, although the performance numbers are not yet as high as expected. Again, multiple optimization opportunities exist. Eventually, an order of magnitude improvement over CPU performance is expected.

6 SUMMARY

We presented a least squares solver implementation for distributed-memory systems with GPU accelerators. The code is part of the SLATE software package, meant to compete with ScaLAPACK and address ScaLAPACK's inability to utilize hardware accelerators. For maximum performance in distributed memory, we applied the communication avoiding QR factorization. For top CPU performance, we implemented blocked and multithreaded panel factorization. GPU acceleration was facilitated by the SLATE software infrastructure, which moves tiles across devices and applies batch matrix multiplication for maximum GPU performance.

SLATE delivers similar performance to ScaLAPACK when using CPUs only, and superior performance when using GPU acceleration. Many improvement opportunities exist—communication, scheduling, further offloading to the GPUs. The main obstacle in achieving GPU saturation is the unforgiving architecture of the target hardware—a massive disproportion of GPU power to CPU power and a huge gap between the computing capabilities of the nodes and the communication capabilities of the network.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of 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.

SOFTWARE

The SLATE software is freely available at https://bitbucket.org/icl/slate. SLATE is distributed under the modified BSD license, imposing minimal restrictions on the use and distribution of the software.

REFERENCES

[1] Bjarne Stig Andersen, John A Gunnels, Fred Gustavson, and Jerzy Wasniewski. 2002. A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. *PARA* 2 (2002), 287–296.

[2] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G Gustavson. 2001. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software (TOMS)* 27, 2 (2001), 214–244.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide*. SIAM.

[4] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and Edgar Solomonik. 2014. Reconstructing Householder vectors from tall-skinny QR. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1159–1170.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.

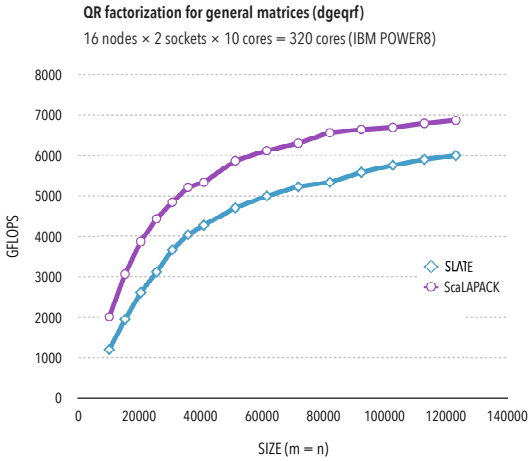

Figure 6: CPU performance of the QR factorization in double precision (`dgeqrt`).

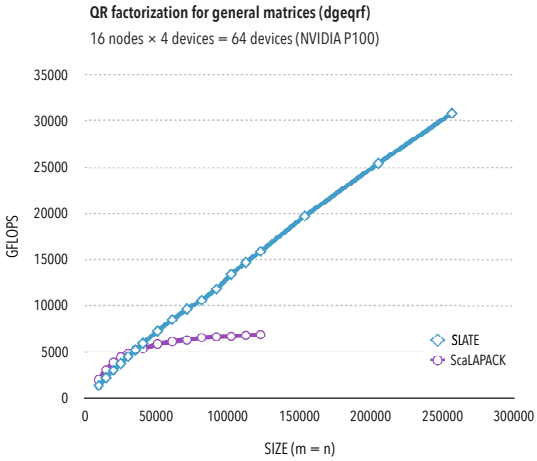

Figure 8: GPU performance of the QR factorization in double precision.

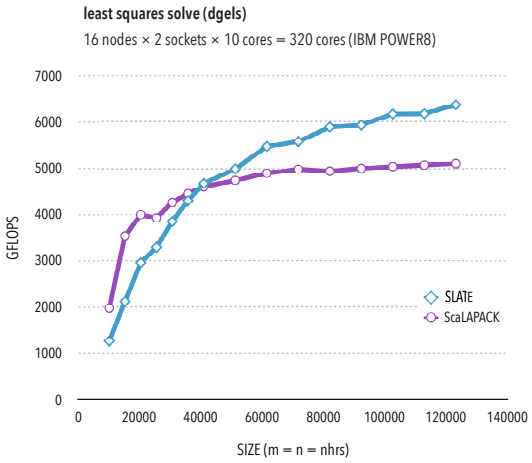

Figure 7: CPU performance of the least square solver in double precision (`dgels`).

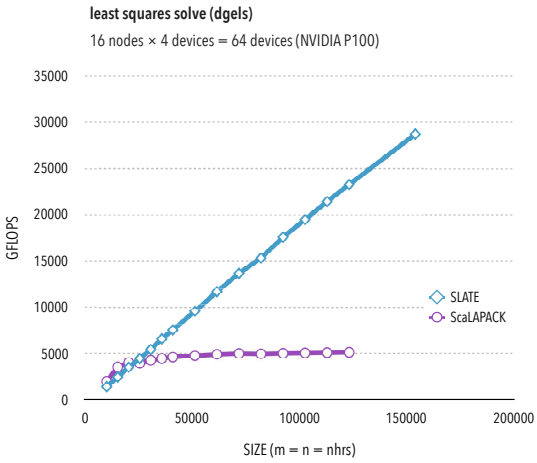

Figure 9: GPU performance of the least square solver in double precision (`dgels`).

[6] Christian Bischof and Charles Van Loan. 1987. The WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.* 8, 1 (1987), s2–s13.

[7] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. 1997. *ScaLAPACK users' guide*. SIAM.

[8] Henricus Bouwmeester, Mathias Jacquelin, Julien Langou, and Yves Robert. 2011. Tiled QR factorization algorithms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 7.

[9] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53.

[10] Anthony Castaldo and Clint Whaley. 2010. Scaling LAPACK panel operations using parallel cache assignment. In *ACM Sigplan Notices*, Vol. 45. ACM, 223–232.

[11] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and Clint Whaley. 1996. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* 5, 3 (1996), 173–184.

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

[13] Jack Dongarra, Mathieu Faverge, Thomas Herault, Mathias Jacquelin, Julien Langou, and Yves Robert. 2013. Hierarchical QR factorization algorithms for multi-core clusters. *Parallel Comput.* 39, 4-5 (2013), 212–232.

[14] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. 2014. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1408–1431.

[15] Erik Elmroth and Fred Gustavson. 1998. New serial and parallel recursive QR factorization algorithms for SMP systems. In *International Workshop on Applied Parallel Computing*. Springer, 120–128.

[16] Erik Elmroth and Fred G Gustavson. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development* 44, 4 (2000), 605–624.

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

[18] Fred Gustavson, André Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. 1998. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Applied Parallel Computing Large Scale Scientific and Industrial Problems* (1998), 195–206.

[19] Fred Gustavson, Lars Karlsson, and Bo Kågström. 2012. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)* 38, 3 (2012), 17.

[20] Randy H Katz, Susan J Eggers, David A Wood, CL Perkins, and Robert G Sheldon. 1985. Implementing a cache consistency protocol. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 276–283.

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

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

[23] Jakub Kurzak, Rajib Nath, Peng Du, and Jack Dongarra. 2010. An implementation of the tile QR factorization for a GPU and multiple CPUs. In *International Workshop on Applied Parallel Computing*. Springer, 248–257.

[24] Robert Schreiber and Charles Van Loan. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Statist. Comput.* 10, 1 (1989), 53–57.

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

[26] Paul Sweazey and Alan Jay Smith. 1986. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News* 14, 2 (1986), 414–423.