Linear Systems Solvers for Distributed-Memory Machines with GPU Accelerators

Jakub Kurzak[1](✉), Mark Gates[1], Ali Charara[1], Asim YarKhan[1], Ichitaro Yamazaki[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, TN 37996, USA
{kurzak,mgates3,charara,yarkhan,iyamazak,dongarra}@icl.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[3] University of Manchester, Manchester M13 9PL, UK
https://www.icl.utk.edu/

Abstract. This work presents two implementations of linear solvers for distributed-memory machines with GPU accelerators—one based on the Cholesky factorization and one based on the LU factorization with partial pivoting. The routines are developed as part of the Software for Linear Algebra Targeting Exascale (SLATE) package, which represents a sharp departure from the traditional conventions established by legacy packages, such as LAPACK and ScaLAPACK. The article lays out the principles of the new approach, discusses the implementation details, and presents the performance results.

Keywords: Linear algebra · Distributed memory · Linear systems of equations · Cholesky factorization · LU factorization · GPU acceleration

1 Introduction

1.1 Linear Systems

Solving a system of linear equations $Ax = b$ is a fundamental capability in scientific and engineering computing. The most common approach is to apply the lower–upper (LU) decomposition, which factors the matrix A as the product of a lower triangular matrix L and an upper triangular matrix U. The procedure usually requires row permutations for numerical stability, referred to as partial pivoting. LU decomposition can be viewed as the matrix form of Gaussian elimination. It is also a key step in inverting a matrix or computing the determinant of a matrix. LU decomposition was introduced by Polish mathematician Tadeusz Banachiewicz.

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).

The system of linear equations $Ax = b$ can be solved much faster when the matrix A is Hermitian, positive definite in complex arithmetic; or symmetric, positive definite in real arithmetics. Commonly, the Cholesky decomposition is used to factor the matrix A into the product of a lower triangular matrix L and its conjugate transpose. It was discovered by a French mathematician, André-Louis Cholesky, for real matrices. When it is applicable, the Cholesky decomposition is roughly twice as efficient as the LU decomposition for solving systems of linear equations.

1.2 SLATE Project

Software for Linear Algebra Targeting Exascale (SLATE)[1] is being developed as part of the Exascale Computing Project (ECP),[2] 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 [3], 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 is 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.

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. Figure 1 shows SLATE in the ECP software stack.

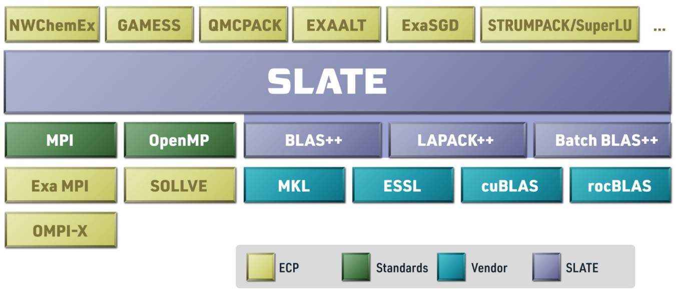

Fig. 1. SLATE in the ECP software stack.

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

2 Motivation

There is an urgent need for multi-GPU accelerated, distributed-memory software. Currently, the fastest machines in United States are the Summit[3] and Sierra[4] systems, at the Oak Ridge National Laboratory (ORNL) and the Lawrence Livermore National Laboratory (LLNL), respectively. As of today, they occupy positions #1 and #2 on the TOP500 list.

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 $22\ (cores) \times 24.56\ gigaFLOP/s = 540.32\ gigaFLOP/s$. The peak performance of the GPUs is $3\ (devices) \times 7.8\ teraFLOP/s = 23.4\ teraFLOP/s$. I.e., 97.7% of performance is on the GPU side, and only 2.3% of performance is on the CPU side.

Also, the U.S. Department of Energy has recently announced plans for achieving exascale. The system, called Frontier, will be built at ORNL. It is planned to go online in 2021 and deliver 1.5 exaFLOP/s of theoretical peak performance. Frontier's nodes will contain one AMD EPYC CPU and four purpose-built AMD Radeon Instinct GPUs.[6]

3 Related Work

Due to the popularity of the Cholesky and LU factorizations, it would be difficult to survey all the related research efforts. Instead we opt for listing the most popular software packages that implement the two routines. Distributed-memory implementations are available in:

– ScaLAPACK (http://www.netlib.org/scalapack/),
– PLAPACK (http://www.cs.utexas.edu/users/plapack/),
– Elemental (http://libelemental.org),
– DPLASMA (http://icl.utk.edu/dplasma/).

While some efforts are being made to GPU-accelerate these packages, at this time we consider these developments experimental. On the other hand, accelerated implementations of the Cholesky and LU factorizations are available in:

– MAGMA (http://icl.cs.utk.edu/magma/),
– CULA (http://www.culatools.com/dense/),
– cuSOLVER (https://developer.nvidia.com/cusolver).

These packages, however, do not support distributed memory. In that respect, the SLATE project seems to be a unique effort in specifically targeting multi-GPU–accelerated distributed-memory systems.

[3] https://www.olcf.ornl.gov/summit/.
[4] https://hpc.llnl.gov/hardware/platforms/sierra.
[5] https://en.wikichip.org/wiki/supercomputers/summit.
[6] https://www.olcf.ornl.gov/frontier/.

4 Original Contribution

This is the only open-source 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. Obviously, very efficient codes were written for the TOP500 runs for these machines. At this point, however, these codes remain proprietary and the details of their inner workings are not publicly available.

The implementations presented here are based on the infrastructure of the SLATE project, which is a radical departure from the established conventions, most notably from the legacy matrix layout of ScaLAPACK. Also, as far as we know, we produced a unique implementation of the LU panel factorization, which combines MPI messaging, OpenMP multithreading, internal blocking, and cache residency.

5 Implementation

5.1 SLATE Basics

Matrix Storage. Unlike legacy dense linear algebra packages, which store the matrix contiguously, by columns, SLATE stores the matrix as a collection of individual tiles. This offers numerous advantages, for example:

- The same structure can be used for holding many different matrix types,[7] 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) [1,2,9] 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,[8] 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 [10].
- Tiles can easily be moved or copied among different memory spaces. Both inter-node communication and intra-node communication are vastly simplified. Tiles can easily and efficiently be transferred between nodes using MPI. Tiles can also be copied to one or more device memories in the case of GPU acceleration.

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

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

In practical terms, the SLATE matrix is implemented by the `std::map` container from the standard C++ library; that is, `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. One issue that may require further attention is the logarithmic complexity of the default implementation of the container in the standard library. If it turns out to be a problem, the use of `std::unordered_map` may be required.

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. SLATE also has the potential to support matrices with non-uniform tile sizes in the future.

For offload to GPU accelerators, SLATE implements a custom memory consistency model, loosely based on the Modified/Owned/Shared/Invalid (MOSI) coherency protocol [13]. The distinguishing feature is that SLATE's model is symmetric; that is, there is no notion of the *main* memory—all memories (host, devices) are considered peers.

Matrix Class Hierarchy. SLATE has the matrix classes below. 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.

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 [8] provides overloaded, precision-independent wrappers for all the underlying LAPACK routines, on which SLATE's least squares routines are built. For instance, `lapack::potrf` in LAPACK++ maps to `spotrf`, `dpotrf`, `cpotrf`, or `zpotrf` 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 quadruple precision (double-double) or half precision (FP16), given appropriate implementations of the elemental operations.

5.2 Cholesky Implementation

SLATE provides routines for solving linear systems of equations, where the coefficient matrix is symmetric (Hermitian) positive definite. These routines compute the factorization $A = LL^T$ ($A = LL^H$) using the Cholesky decomposition, and follow with the steps of forward and backward substitution. The routines are mathematically equivalent to their ScaLAPACK counterparts [6].

Figure 2 (left picture) shows the basic mechanics of the Cholesky factorization in SLATE. Like most routines in SLATE, the implementation relies on nested tasking using the OpenMP standard, with the top level responsible for scheduling a small number of coarse-grained, interdependent tasks, and the nested level 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 Basic Linear Algebra Subprograms (BLAS) routines, to exploit the efficiency of processing large numbers of tiles in one call to a GPU kernel.

The Cholesky factorization in SLATE applies the technique of *lookahead* [5,11,14], 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.

5.3 LU Implementation

SLATE provides routines for solving linear systems of equations, where the coefficient matrix is a general (nonsymmetric) matrix. These routines compute the factorization $PA = LU$ using the process of Gaussian elimination with partial

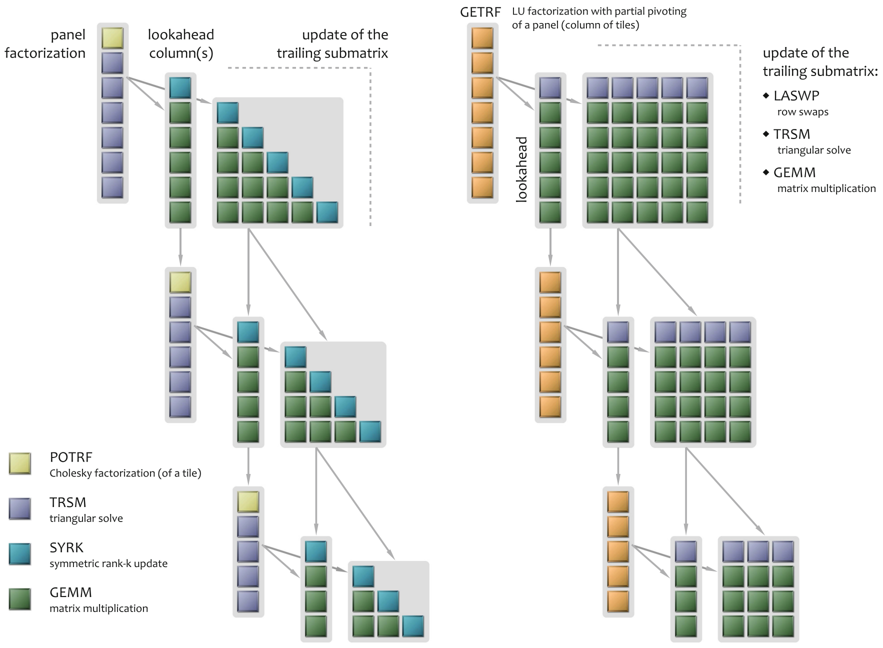

Fig. 2. Left: Cholesky factorization with lookahead of one. Right: LU factorization with lookahead of one.

(row) pivoting, and follow with the steps of forward and backward substitution. The routines are mathematically equivalent to their ScaLAPACK counterparts [6].

Figure 2 (right picture) shows the basic mechanics of the LU factorization in SLATE. While the parallelization is based on the same principles as the Cholesky factorization, the implementation is significantly more challenging, due to the application of row pivoting. The primary consequence of row pivoting is a fairly complex, and heavily synchronous, panel factorization procedure. The secondary effect is the communication overhead of swapping rows to the left and to the right of the panel. A further complication is introduced by GPU acceleration, which requires layout translation, as the row swapping operation is extremely inefficient in column major.

The critical component of the LU factorization is the step of factoring the panel, which in SLATE is an arbitrary selection of tiles from one column of the matrix. This operation is on the critical path of the algorithms and has to be optimized to the maximum. Resorting to a simple, memory-bound implementation could have profoundly negative consequences for performance. The current implementation of the LU panel factorization in SLATE is derived from the technique of *Parallel Cache Assignment* (PCA) by Castaldo et al. [4], and the work on parallel panel factorization by Dongarra et al. [7].

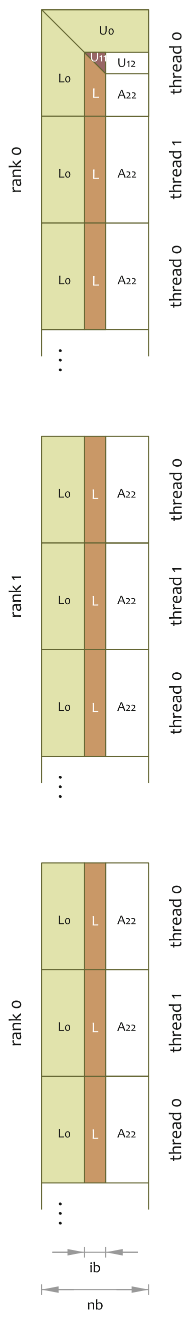

Fig. 3. LU panel.

The LU panel factorization in SLATE relies on internal blocking and persistent assignment of tiles to threads within each MPI process. Unlike past implementations, it is not recursive, as plain recursion proved inferior to blocking. Memory residency provides some level of cache reuse, while blocking provides some level of compute intensity. The resulting implementation is no longer memory bound, and scales well with the number of processes and the number of threads in each process. The procedure is heavily synchronous and relies on MPI collective communication to exchange pivot information, and on thread barriers for intra-node synchronization. An MPI sub-communicator is created for each set of processes participating in each panel factorization.

Figure 3 shows the basic premise of the panel implementation. The tiles are assigned to MPI ranks, and to threads within each rank, in a round-robin fashion. The assignment is persistent, which allows for a high degree of cache reuse, within each rank, throughout the panel factorization. Also, the routine is internally blocked: 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 level 1 and 2 BLAS operations, but can benefit to some extent from cache residency, while the *nb* factorization contains mostly level 3 BLAS operations and can also benefit from cache residency.

At each step of the *ib* panel factorization, a stripe of the lower triangular matrix (L) is computed, along with a small part of the U factor (U_{11}). All this work is done one column at a time. What follows is application of the L transformations to the right, which includes updating the remaining A_{22} submatrix, and computing of a new horizontal stripe of the U factor (U_{12}). Most of this work is done using level 3 BLAS operations.

Each panel factorization is followed by an update of the trailing submatrix (Fig. 2), which involves: (1) applying row swaps (`laswp`), (2) triangular solve (`trsm`), and (3) matrix multiplication (`gemm`). This requires the following communication: (1) "horizontal" broadcasting of the panel to the right, (2) "vertical" exchanges of the rows being swapped, and (3) "vertical" broadcasting of the top row or tiles down the matrix.

This creates the extra complication of multiple OpenMP tasks issuing, possibly concurrently, independent communications. Specifically, the collective communication of the panel factorization may coincide with sends and receives of multiple simultaneous row swaps. This requires that the underlying MPI implementation be thread safe, and support the `MPI_THREAD_MULTIPLE` mode (i.e., multiple threads simultaneously issuing MPI communications). It also requires that the different communications be distinguished by different MPI tags.

6 Results

6.1 Setup

Performance numbers were collected using the SummitDev system[9] at the Oak Ridge Leadership Computing Facility (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.

6.2 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. ScaLAPACK was run with one process per core, which is still the prevailing method of getting the best performance from ScaLAPACK. SLATE, on the other hand, was run using one process per GPU. While SLATE does provide multi-GPU support, the best performance was reached by assigning each GPU to one process and splitting the CPU cores evenly (i.e., five cores per process).

Figure 4 shows performance comparison of SLATE and ScaLAPACK for the Cholesky factorization. The left chart shows performance when using CPUs only for both SLATE and ScaLAPACK. The right chart compares CPU performance of ScaLAPACK with GPU performance of SLATE. At this point, we are not aware of an efficient way of GPU-accelerating ScaLAPACK.

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

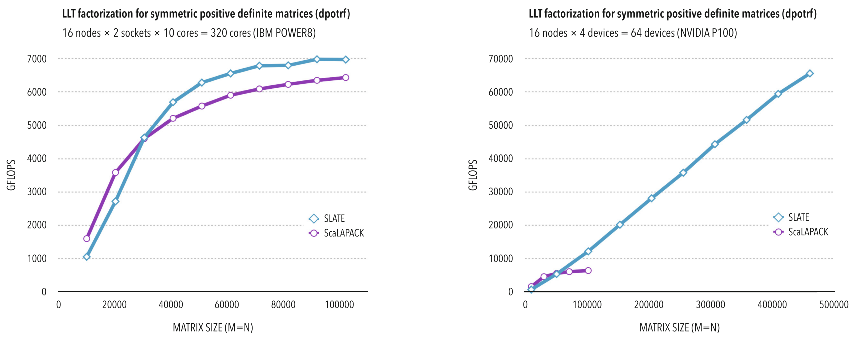

Fig. 4. Performance of `dpotrf` without acceleration (left) and with acceleration (right). The CPU peak is 8,960 gigaFLOPs, the GPU peak is 339,200 gigaFLOPs.

Similarly, Fig. 5 shows a performance comparison of SLATE and ScaLAPACK for the LU factorization. The left chart shows performance when using CPUs only for both SLATE and ScaLAPACK. The right chart compares CPU performance of ScaLAPACK with GPU performance of SLATE.

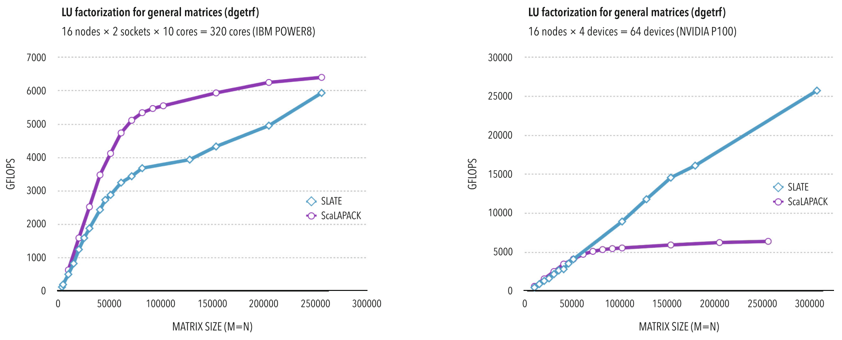

Fig. 5. Performance of `dgetrf` without acceleration (left) and with acceleration (right). The CPU peak is 8,960 gigaFLOPs, the GPU peak is 339,200 gigaFLOPs.

6.3 Discussion

For the Cholesky factorization, SLATE delivers superior performance compared to ScaLAPACK. The CPU performance of SLATE is higher than the CPU performance of ScaLAPACK, and SLATE delivers an order of magnitude speedup from GPU acceleration. For the LU factorization, the CPU performance of SLATE is lower than ScaLAPACK's for smaller matrix sizes, but catches up for larger sizes. GPU performance of LU is generally superior to ScaLAPACK's, although the gains of acceleration are smaller than for Cholesky.

While SLATE clearly benefits from GPU acceleration, it only achieves a small fraction of the GPU theoretical peak performance. This is mostly due to the fact that the computing power of the GPUs completely outmatches the communication capabilities of the interconnection, despite the fact that the network represents state-of-the-art technology. With this trend continuing, it will be necessary to seek new algorithms—algorithms that are even more compute-intensive than the traditional solutions to dense linear algebra problems. One such example is the QDWH algorithm [15] for computing the singular value decomposition (SVD).

Another problem is the one of mixing MPI messaging with OpenMP multithreading. In SLATE, MPI messages are sent from inside OpenMP tasks, which requires the highest level of MPI thread safety (`MPI_THREAD_MULTIPLE`) and some other precautions to prevent deadlock. These measures have an adverse effect on performance. Ultimately, what is needed is an `MPI_TASK_MULTIPLE` mode of operation, as described by Sala et al. [12].

Finally, the biggest factor contributing to the poor performance of the LU factorization is the cost of pivoting (i.e., the operation of swapping rows). Currently, it is done in a sequential fashion, the same way it is done in LAPACK and ScaLAPACK. Moving to parallel pivoting, where all the rows can be swapped simultaneously, may improve the situation. Also, storing the matrix in column-major in the CPU memory has a significant impact on the performance of pivoting on the CPU side, and moving the CPU operations to row-major—same as was done for GPUs—may be necessary.

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.

Software. The SLATE software if 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. Andersen, B.S., Gunnels, J.A., Gustavson, F., Wasniewski, J.: A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. PARA **2**, 287–296 (2002)
2. Andersen, B.S., Waśniewski, J., Gustavson, F.G.: A recursive formulation of Cholesky factorization of a matrix in packed storage. ACM Trans. Math. Softw. (TOMS) **27**(2), 214–244 (2001)
3. Blackford, L.S., et al.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
4. Castaldo, A., Whaley, C.: Scaling LAPACK panel operations using parallel cache assignment. In: ACM Sigplan Notices, vol. 45, pp. 223–232. ACM (2010)

5. Chan, E., van de Geijn, R., Chapman, A.: Managing the complexity of lookahead for LU factorization with pivoting. In: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 200–208. ACM (2010)
6. Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., Whaley, C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Sci. Program. **5**(3), 173–184 (1996)
7. Dongarra, J., Faverge, M., Ltaief, H., Luszczek, P.: Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. Concurr. Comput. Pract. Exp. **26**(7), 1408–1431 (2014)
8. Gates, M., et al.: 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
9. Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B., Ling, P.: Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In: Kågström, B., Dongarra, J., Elmroth, E., Waśniewski, J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 195–206. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0095337
10. Gustavson, F., Karlsson, L., Kågström, B.: Parallel and cache-efficient in-place matrix storage format conversion. ACM Trans. Math. Softw. (TOMS) **38**(3), 17 (2012)
11. Kurzak, J., Dongarra, J.: Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 147–156. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75755-9_18
12. Sala, K., Teruel, X., Perez, J.M., Peña, A.J., Beltran, V., Labarta, J.: Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Comput. **85**, 153–166 (2019)
13. Sorin, D.J., Hill, M.D., Wood, D.A.: A primer on memory consistency and cache coherence. Synth. Lect. Comput. Arch. **6**(3), 1–212 (2011)
14. Strazdins, P., et al.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization (1998)
15. Sukkari, D., Ltaief, H., Keyes, D.: A high performance QDWH-SVD solver using hardware accelerators. ACM Trans. Math. Softw. (TOMS) **43**(1), 6 (2016)