SLATE WORKING NOTES 12

Implementing Matrix Inversions

Jakub Kurzak
Mark Gates
Ali Charara
Asim YarKhan
Jack Dongarra

Innovative Computing Laboratory

August 13, 2019

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

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

Revision	Notes
06-2019	first publication
07-2019	technical editing
08-2019	performance charts updated with ScaLAPACK performance numbers

```
@techreport{kurzak2019implementing,
  author={Kurzak, Jakub and Gates, Mark and Charara, Ali and
          YarKhan, Asim and Dongarra, Jack},
  title={{SLATE} Working Note 12: Implementing Matrix Inversions},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2019},
  month={June},
  number={ICL-UT-19-04},
  note={revision 07-2019}
}
```

Contents

List of Figures

CHAPTER 1

Design of SLATE

Software for Linear Algebra Targeting Exascale (SLATE) [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). SLATE will deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems, including GPU-accelerated systems as well as more traditional multi core–only systems. SLATE will provide coverage of existing ScaLAPACK functionality, including parallel implementations of Basic Linear Algebra Subroutines (BLAS), linear systems solvers, least squares solvers, and singular value and eigenvalue solvers. In this respect, SLATE will serve as a replacement for ScaLAPACK, which, after two decades of operation, cannot be adequately retrofitted for modern, GPU-accelerated architectures. Figure 1.1 shows SLATE's position in the ECP software stack. SLATE will accomplish its goals by using modern techniques like tiled matrix storage, task-based dynamic scheduling, and communication-avoiding algorithms, along with a modern C++ framework, briefly discussed in the following paragraphs.

Tiled Matrix Layout: The new matrix storage introduced in SLATE is one of its most impactful features. In this respect, SLATE represents a radical departure from other distributed linear algebra software such as ScaLAPACK or Elemental, where the local matrix occupies a contiguous memory region on each process. In contrast, tiles are first class objects in SLATE that can be individually allocated and passed to low-level tile routines. In SLATE, the matrix consists of a collection of individual tiles with no correlation between their positions in the matrix and their memory locations. At the same time, SLATE also supports tiles pointing to data in a traditional ScaLAPACK matrix layout, thereby easing an application's transition from ScaLAPACK to SLATE.

[1] http://icl.utk.edu/slate/

[2] https://www.exascaleproject.org

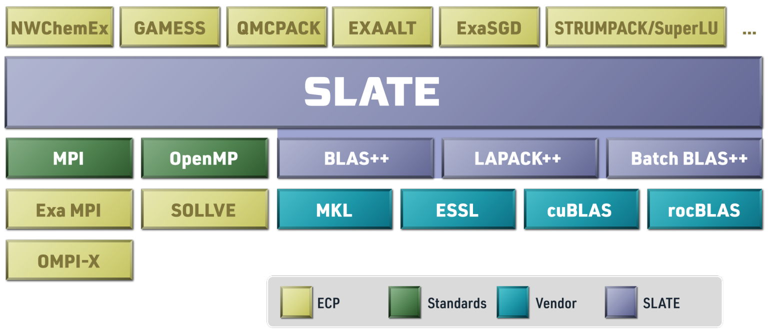


Figure 1.1: SLATE in the ECP software stack.

Object-Oriented Design: The design of SLATE revolves around the Tile class and the Matrix class hierarchy. The Tile class is intended as a simple class for maintaining the properties of individual tiles and implementing core serial tile operations, such as tile BLAS, while the Matrix class hierarchy maintains the state of distributed matrices throughout the execution of parallel matrix algorithms in a distributed-memory environment. Currently, the classes are structured as follows:

- **BaseMatrix** is an abstract base class for all matrices.
  - **Matrix** represents a general $m \times n$ matrix.
  - **BaseTrapezoidMatrix** is an abstract base class for all upper-trapezoid or lower-trapezoid, $m \times n$ matrices. For upper matrices, tiles $A(i, j)$ are stored for $i \leq j$. For lower matrices, tiles $A(i, j)$ are stored for $i \geq j$.
    - **TrapezoidMatrix** represents an upper-trapezoid or a lower-trapezoid, $m \times n$ matrix. The opposite triangle is implicitly zero.
      - **TriangularMatrix** represents an upper-triangular or a lower-triangular, $n \times n$ matrix.
    - **SymmetricMatrix** represents a symmetric, $n \times n$ matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry ($A_{j,i} = A_{i,j}$).
    - **HermitianMatrix** represents a Hermitian, $n \times n$ matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry ($A_{j,i} = \bar{A}_{i,j}$).

Handling of side, uplo, trans: The classical BLAS takes parameters such as `side`, `uplo`, `trans` (named "`op`" in SLATE), and `diag` to specify operation variants. Traditionally, this has meant that implementations have numerous cases. The reference BLAS has nine cases in `zgemm` and eight cases in `ztrmm` (times several sub-cases). ScaLAPACK and PLASMA likewise have eight cases in `ztrmm`. In contrast, by storing both `uplo` and `op` within the matrix object itself, and supporting inexpensive shallow copy transposition, SLATE can implement just one or two cases and map all the other cases to that implementation by appropriate transpositions. For instance, SLATE only implements one case for `gemm` (`NoTrans, NoTrans`) and handles all other cases by swapping indices of tiles and setting `trans` appropriately for the underlying tile operations.

Dynamic Scheduling: Dataflow scheduling (`omp task depend`) is used to execute a task graph with nodes corresponding to large blocks of the matrix. Dependencies are tracked using dummy vectors, where each element represents a block of the matrix, rather than the matrix data itself. For multi-core execution, each large block is dispatched to multiple cores—using either nested tasking (`omp task`) or batched BLAS. For GPU execution, calls to batched BLAS are used specifically to deliver fast processing of matrix blocks that are represented as large collections of tiles.

Templating of Execution Targets: Parallelism is expressed in SLATE's computational routines. Each computational routine solves a sub-problem, such as computing an LU factorization (`getrf`) or solving a linear system given an LU factorization (`getrs`). In SLATE, these routines are templated for different targets (CPU or GPU), with the code typically independent of the target. The user can choose among various target implementations:

Target::HostTask means multithreaded execution by a set of OpenMP tasks.

Target::HostNest means multithreaded execution by a nested "`parallel for`" loop.

Target::HostBatch means multithreaded execution by calling a batched BLAS routine.

Target::Devices means (multi-)GPU execution using calls to batched BLAS.

MPI 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 (sub)matrix to be updated. This way, SLATE's messaging layer is oblivious to the mapping of tiles to processes. Also, multiple broadcasts are aggregated to allow for pipelining of MPI messages with transfers between the host and the devices. Since the set of processes involved in a broadcast is determined dynamically, the use of MPI collectives is not ideal, as it would require setting up a new subcommunicator for each broadcast. Instead, SLATE uses point-to-point MPI communication following a hypercube pattern to broadcast the data.

Node-Level Coherency: For offload to GPU accelerators, SLATE implements a memory consistency model, inspired by the MOSI cache coherency protocol [1, 2], 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. A tile's instance can be in one of three states: *Modified*, *Shared*, or *Invalid*. Additional flag *OnHold* can be set along any state, as follows:

Modified (M) indicates that the tile's data is modified. Other instances should be *Invalid*. The instance cannot be purged.

Shared (S) indicates that the tile's data is up-to-date. Other instances may be *Shared* or *Invalid*. The instance may be purged unless it is on hold.

Invalid (I) indicates that the tile's data is obsolete. Other instances may be *Modified*, *Shared*, or *Invalid*. The instance may be purged unless it is on hold.

OnHold (O) is a flag orthogonal to the other three states that indicates a hold is set on the tile instance, and the instance cannot be purged until the hold is released.

Templating of 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. Operations are defined so that they can be applied consistently across all precisions. SLATE's BLAS++ component provides overloaded, precision-independent wrappers for all underlying, node-level BLAS, and SLATE's PBLAS are built on top of these. Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE code should be able to accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS.

CHAPTER 2

Implementation of Inversions

In linear algebra, it is ill-advised to invert a matrix in order to solve a system of linear equations, $Ax = B$. Instead, the matrix is factored into a product of a lower-triangular matrix and an upper-triangular matrix, and the system is solved using a forward substitution and a backward substitution. However, explicit construction of the inverse A^{-1} is an important tool in probability theory and statistics with practical applications in computational chemistry and material science.

One of the fundamental concepts in statistics is that of the *covariance matrix*. A covariance matrix, also known as an *auto-covariance matrix*, a *dispersion matrix*, a *variance matrix*, or a *variance–covariance matrix*, is a matrix where the element in the i, j position is the covariance between the i-th and j-th elements of a random vector. Intuitively, the covariance matrix generalizes the notion of *variance* to multiple dimensions. In statistics, *precision* is the reciprocal of the *variance*, and the *precision matrix* (also known as a *concentration matrix*) is the matrix inverse of the covariance matrix. The precision matrix is a measure of how tightly clustered the variables are around the mean (the diagonal elements) and the extent to which they do not co-vary with the other variables (the off-diagonal elements).

The numerical stability of matrix inversion algorithms was analyzed by Du Croz and Higham [3]. Quintana et al. [4] and Bientinesi et al. [5] discussed parallel implementations of matrix inversions for shared-memory and distributed-memory computers. Agullo et al. [6] and Bouwmeester and Langou [7] discussed dataflow scheduling of Cholesky-based matrix inversion algorithms with emphasis on the analysis of the critical path. Dongarra et al. [8] presented a similar study for LU-based inversion.

2.1 Cholesky-Based Inversion

Inversion of a symmetric positive–definite matrix is done in place with the result overriding the original matrix. The operation proceeds in three stages, as shown on Figure 3.1. First, the matrix is factored into a product of a lower-triangular matrix and its conjugate transpose, $A = LL^H$. Then, the inverse of L is computed and subsequently multiplied by its transpose.

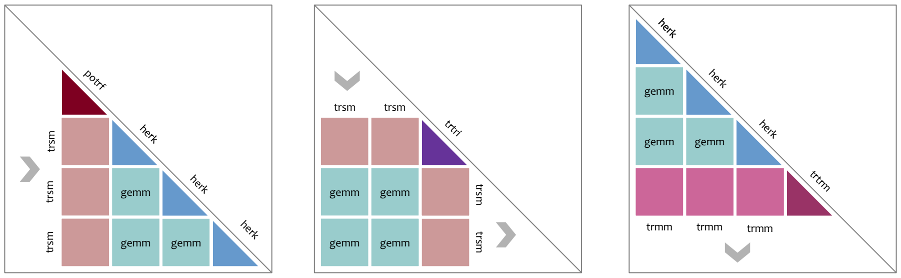


Figure 2.1: Stages of lower-triangular Cholesky-based inversion.

The first stage of the inversion is the Cholesky factorization of the input matrix (Figure 2.2). Step (1) computes the Cholesky factorization (`potrf`) of the diagonal tile, step (2) applies the triangular solve (`trsm`) to the tiles below, and step (3) applies updates to the trailing submatrix. Step (3) is a `syrk`/`herk` operation, which internally translates to `syrk`/`herk` on the diagonal tiles and `gemm` on the off-diagonal tiles. The right-looking Cholesky factorization is a very efficient operation, as it exposes a large number of `gemm` calls and enables the application of lookahead.

(1) $A_{k,k} = LL^T$

(2) $A_{k+1:nt-1,k} = A_{k+1:nt-1,k}/A_{k,k}$

(3) $A_{k+1:nt-1,k+1:nt-1} = A_{k+1:nt-1,k+1:nt-1} - A_{k+1:nt-1,k} * A^H_{k+1:nt-1,k}$

Figure 2.2: Operations of the Cholesky factorization.

The second stage is the triangular inversion of the L factor (Figure 2.3). Step (1) applies the triangular solve (`trsm`) to the leading column of tiles, step (2) updates the rectangular off-diagonal block with the product of the leading column of tiles and the trailing row of tiles (`gemm`), step (3) applies triangular solve (`trsm`) to the trailing row, and step (4) inverts the diagonal tile (`trtri`). The triangular inversion is also a very efficient operation, as it also exposes a large number of `gemm` calls and enables the application of lookahead.

(1) $A_{k+1:nt-1,k} = A_{k+1:nt-1,k}/A_{k,k}$

(2) $A_{k+1:nt-1,0:k-1} = A_{k+1:nt-1,0:k-1} + A_{k+1:nt-1,k} * A_{k,0:k-1}$

(3) $solve\ A_{k,k} * A_{k,0:k-1} = A_{k,0:k-1}$

(4) $A_{k,k} = A^{-H}_{k,k}$

Figure 2.3: Operations of triangular inversion.

The third stage is the in-place multiplication of a triangular matrix by its transpose. Step (1) is a `syrk`/`herk` operation, which translates to `syrk`/`herk` on the diagonal tiles and `gemm` on the off-diagonal tiles, step (2) is a multiplication of the leading row of tiles by the triangular tile on the diagonal (`trmm`), and step (3) is a multiplication of the diagonal tile by its transpose. This operation is called `lauum` in LAPACK but was renamed as `trtrm` in SLATE. This stage is not as efficient as the other stages. Although it also exposes a large number of `gemm` calls, lookahead is not possible due to the *write after read* dependency on the leading row.

(1) $A_{0:k-1,0:k-1} = A_{0:k-1,0:k-1} + A^H_{k,0:k-1} * A_{k,0:k-1}$

(2) $A_{k,0:k-1} = A_{k,0:k-1} * A^H_{k,k}$

(3) $A_{k,k} = A^H_{k,k} * A_{k,k}$

Figure 2.4: Operations of in-place triangular multiplication.

The last stage is the only operation that schedules poorly. It could be dramatically improved if implemented out-of-place, as shown by Bouwmeester and Langou [7]. However, as shown in the same article, there is a better alternative. In fact, the three stages can be pipelined very well. As `potrf` moves to the right, `trtri` can proceed to the left, while `trtrm` can ascend from the top. In that case, `trtrm` can start executing almost immediately and execute in parallel with the other two stages.

This pipelining occurs naturally if the computation is expressed as a direct acyclic graph (DAG), which is the case in the PLASMA library. It is, however, not the case in SLATE, which does not rely on dynamic scheduling to the extent that PLASMA does. Currently, the stages are implemented as three separate routines and not pipelined. The fully pipelined version of the Cholesky-bases inversion will be the target of future efforts.

2.2 LU-Based Inversion

In LAPACK and ScaLAPACK, the LU-based inversion follows the same principle of executing in-place (i.e., overriding the input matrix with the inverse). The same approach was taken in SLATE to produce the CPU implementation. However, the in-place algorithm turned out to be difficult to GPU accelerate, and an out-of-place routine was implemented for GPUs. Here we describe the canonical, in-place code first.

The in-place, LU-based inversion proceeds in three main stages and is followed by column pivoting at the end. The first stage is computing the LU factorization with partial (row) pivoting of the input matrix, $A = PLU$, and the second stage is computing the triangular inverse of U (Figure 2.5). The triangular inverse operation is identical to the one described previously in the Cholesky section, except here it is executed on the upper-triangular matrix U, not the lower-triangular matrix, L. In SLATE, both are implemented by the same piece of code by the using techniques described in Chapter 1.

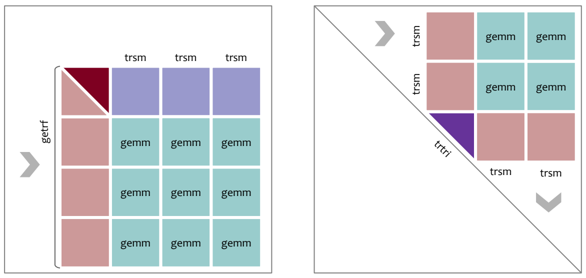


Figure 2.5: The first two stages of LU-based inversion.

The first stage of the inversion is the LU factorization of the input matrix (Figure 2.6). Step (1) computes the LU factorization (`getrf`) of a column (panel) of tiles. Step (2) applies the resulting row permutations to the trailing submatrix. Step (3) applies the permutations to the left of the panel (currently postponed to the end of the factorization). Step (4) performs the triangular solve (`trsm`) on the top row, and step (5) updates the block below with the product (`gemm`) of the panel column (minus the diagonal block) and the top row. The LU factorization exposes a large number of `gemm` calls and would be a very efficient workload if it was not for the impact of pivoting. In reality, LU factorization performs poorly due to the cost of row swaps, which are very inefficient in a GPU-accelerated, distributed-memory environment.

(1) $A_{k:nt-1,k} = PLU$

(2) $A_{k:nt-1,k+1:nt-1} = P * A_{k:nt-1,k+1:nt-1}$

(3) $A_{k:nt-1,0:k-1} = P * A_{k:nt-1,0:k-1}$

(4) $solve\ A_{k,k} A_{k,k+1:nt-1} = A_{k,k+1:nt-1}$

(5) $A_{k+1:nt-1,k+1:nt-1} = A_{k+1:nt-1,k+1:nt-1} - A(k+1 : nt-1, k) * A(k, k+1 : nt-1)$

Figure 2.6: Operations of the LU factorization.

The third stage is the backward substitution (Figures 2.8 and 3.2). In this stage, the matrix is swept from right to left. Step (1) copies a (lower-trapezoid) column k of the matrix L to a workspace and zeroes its original location. Then, step (2) multiplies (`gemm`) the matrix A to the right of k by the rectangular part of the workspace and accumulates the result in the column k of A. Finally, step (3) performs a triangular solve (`trsm`) in the trapezoid column k of L by applying the diagonal triangle to the tiles below.

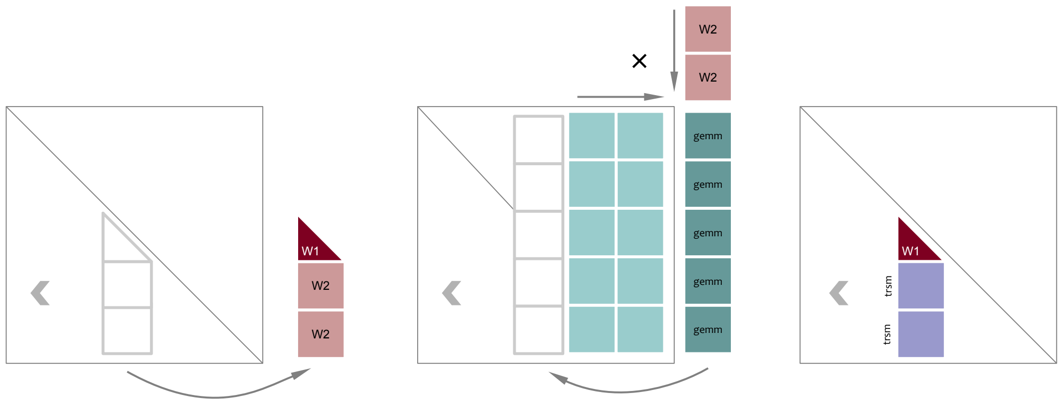


Figure 2.7: The last stage of LU-based inversion.

While the stage of backward substitution exposes a large number of `gemm` calls, it suffers from two critical performance issues. First, it sweeps from left to right and has a *read after write* dependency on the leading column, which prevents lookahead. Second, the output of the `gemm` operation is a tall and skinny matrix, which makes this operation more difficult to parallelize (efficient implementation requires computation of partial sums followed by a reduction).

(1) $L_{k,k} \rightarrow W_1$
$L_{k,k} = 0$
$L_{k+1:A_nt-1,k} \rightarrow W_2$
$L_{k+1:A_nt-1,k} = 0$

(2) $A_{:,k} = A_{:,k} - A_{:,k+1:nt-1} * W$

(3) $A_{k+1:nt-1,k} = A_{k+1:nt-1,k} / A_{k,k}$

Figure 2.8: Operations of the last stage of LU-based inversion.

The in-place, LU-based inversion is also hindered by two more serious performance issues. First, because the last stage (backward substitution) sweeps the matrix from right to left, it cannot be pipelined with the previous stages (as discussed by Dongarra et al. [8]). Second, column pivoting is required at the end. The problem is that column pivoting can only be done efficiently in column-major layout—especially when GPUs are involved. At the same time, the first stage (LU factorization) performs row pivoting, which requires a row-major layout for efficiency. So, we can only—efficiently—do one or the other, or we can apply layout translation in between, none of which are good options.

All of these challenges make the in-place, LU-based inversion a bad candidate for GPU acceleration. With little hope of getting this implementation accelerated, we provided a straightforward out-of-place implementation. The routine has an additional parameter for passing a separate output matrix for storing the result. Thanks to overloading, the routine has the same name (`getrf`), and only the signature is different (Chapter 4).

In the out-of-place implementation, the output matrix is simply initialized with the identity, and the forward and backward substitutions are done by applying row pivoting and two subsequent triangular solves (`trsm`). There are two performance benefits. First, triangular solves are intensive in `gemm` calls and offload well to accelerators. Second, both the factorization and the forward substitution apply row pivoting, so both operations can be done efficiently after the matrix is translated to row major.

The downside is the increased number of floating-point operations. The cost of the in-place inversion (the `getri` routine) is $4/3N^3$, while the cost of two triangular solves is $2N^3$. At this time, though, it still seems to be the only feasible way to GPU accelerate the operation.

CHAPTER 3

Performance of Inversions

3.1 Environment

Performance numbers were collected using the Summit system [1,2] at the Oak Ridge Leadership Computing Facility (OLCF). Summit is equipped with IBM POWER9 processors and NVIDIA V100 (Volta) GPUs. Each of Summit's nodes contains two POWER9 CPUs (with 22 cores each) and six V100 GPUs. Each node has 512 GB of DDR4 memory, and each GPU has 16 GB of HBM2 memory. NVLink 2.0 provides all-to-all 100 GB/s connections for one CPU and three GPUs (i.e., one CPU is connected to three GPUs with 100 GB/s bandwidth each, and each GPU is connected to the other two with 100 GB/s bandwidth each). The two CPUs are connected with a 64 GB/s X Bus. Each node has a Mellanox enhanced-data rate (EDR) InfiniBand network interface controller (NIC) that supports 100 Gbps of bi-directional traffic.

The software environment used for the experiments included:

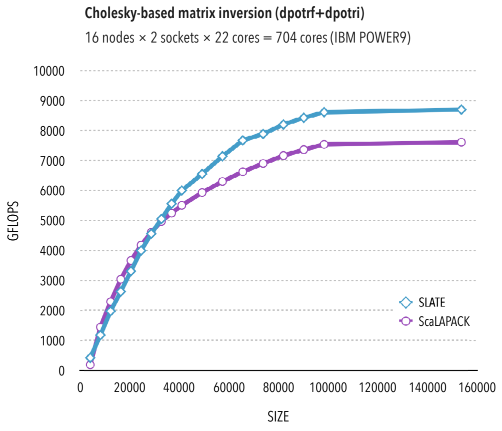
- GNU Compiler Collection (GCC) 6.4.0,
- CUDA 10.1.105,
- Engineering Scientific Subroutine Library (ESSL) 6.1.0,
- Spectrum MPI 10.3.0.0,
- Netlib LAPACK 3.8.0, and
- Netlib ScaLAPACK 2.0.2.

[1]https://www.olcf.ornl.gov/summit/
[2]https://en.wikichip.org/wiki/supercomputers/olcf-4

3.2 Performance

All runs were performed using 16 nodes of the Summit system, which provide $16\ nodes \times 2\ sockets \times 22\ cores = 704$ IBM POWER9 cores and $16\ nodes \times 4\ devices = 96$ NVIDIA V100 accelerators. The runs were done using one process per CPU socket. I.e., one process had 22 cores and 3 devices at its disposal.

Figure 3.1 shows the performance of the Cholesky-based inversion (the in-place implementation). In CPU-only runs, SLATE is $\sim 20\%$ faster than ScaLAPACK for matrices of size 150,000. GPU-accelerated SLATE has $\sim 9\times$ performance advantage over unaccelerated ScaLAPACK for matrices of size $350,000$.



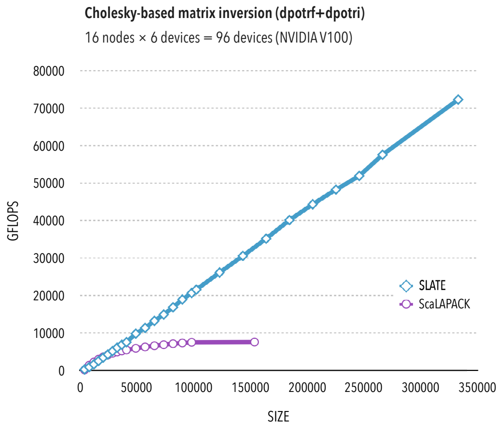
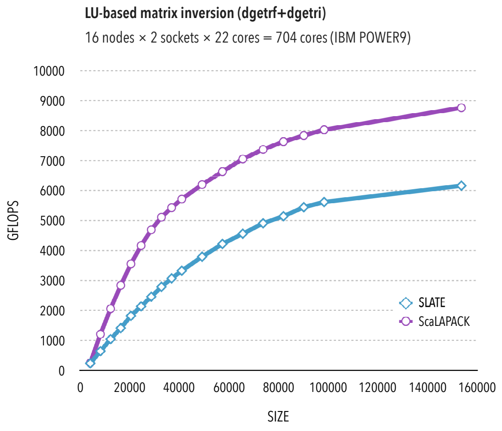


Figure 3.1: Performance of Cholesky-based inversion.

Figure 3.2 shows the performance of the LU-based inversion (the out-of-place implementation). In CPU-only runs, SLATE is $\sim 30\%$ slower than ScaLAPACK for matrices of size 150,000. GPU-accelerated SLATE has $\sim 3.7\times$ performance advantage over unaccelerated ScaLAPACK for matrices of size $350,000$.



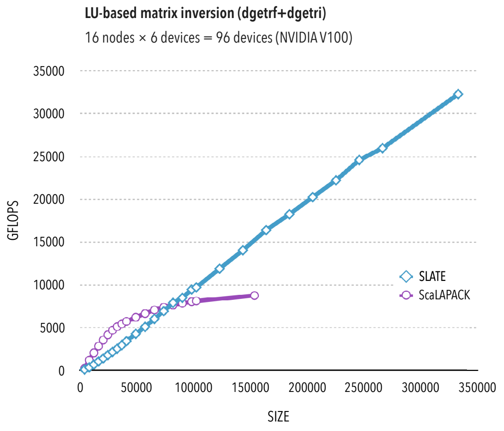


Figure 3.2: Performance of LU-based inversion.

The Cholesky-based inversion performs substantially better than the LU-based inversion. The CPU code achieves a bigger fraction of the peak CPU performance, and the GPU code provides substantial acceleration. Further performance improvement is possible by pipelining the three stages of the workload: the Cholesky factorization (`potrf`), the triangular inversion (`trtri`), and the triangular matrix multiplication (`trtrm`).

The LU-based inversion performs much worse. The CPU code achieves a smaller fraction of the CPU peak, and the benefit of acceleration is also smaller. The LU factorization suffers a from the overhead of row pivoting, and the out-of-place implementation performs substantially more floating-point operations. Further performance improvement is possible by optimizing the process of pivoting and by pipelining the factorization and the triangular inversion.

CHAPTER 4

Function Signatures

The Cholesky-based inversion is accomplished by calling the `potrf` routine and then the `potri` routine. The routines' signatures are as follows.

```
template <typename scalar_t>
void potrf(HermitianMatrix<scalar_t>& A,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void potri(HermitianMatrix<scalar_t>& A,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

The LU-based inversion is accomplished by calling the `getrf` routine and then one of the `getri` routines—either in-place (CPU-only) or out-of-place (CPU or GPU). The routines' signatures are as follows.

```
template <typename scalar_t>
void getrf(Matrix<scalar_t>& A, Pivots& pivots,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void getri(Matrix<scalar_t>& A, Pivots& pivots,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void getri(Matrix<scalar_t>& A, Pivots& pivots,
           Matrix<scalar_t>& B,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

Bibliography

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

[2] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[3] Jeremy J. Du Croz and Nicholas J. Higham. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12(1):1–19, 1992.

[4] Enrique S Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.

[5] Paolo Bientinesi, Brian Gunter, and Robert A Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):3, 2008.

[6] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *International Conference on High Performance Computing for Computational Science*, pages 129–138. Springer, 2010.

[7] Henricus Bouwmeester and Julien Langou. A critical path approach to analyzing parallelism of algorithmic variants. Application to Cholesky inversion. *arXiv preprint arXiv:1010.2000*, 2010.

[8] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, pages 33–42. ACM, 2011.