SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library

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

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

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

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

Jack Dongarra
dongarra@icl.utk.edu
University of Tennessee
Knoxville, TN
Oak Ridge National Laboratory
University of Manchester

ABSTRACT

The SLATE (Software for Linear Algebra Targeting Exascale) library is being developed to provide fundamental dense linear algebra capabilities for current and upcoming distributed high-performance systems, both accelerated CPU–GPU based and CPU based. SLATE will provide coverage of existing ScaLAPACK functionality, including the parallel BLAS; linear systems using LU and Cholesky; least squares problems using QR; and eigenvalue and singular value problems. In this respect, it will serve as a replacement for ScaLAPACK, which after two decades of operation, cannot adequately be retrofitted for modern accelerated architectures. SLATE uses modern techniques such as communication-avoiding algorithms, lookahead panels to overlap communication and computation, and task-based scheduling, along with a modern C++ framework. Here we present the design of SLATE and initial reports of several of its components.

CCS CONCEPTS

• **Mathematics of computing** → **Computations on matrices**; • **Computing methodologies** → *Massively parallel algorithms*; *Shared memory algorithms*;

KEYWORDS

dense linear algebra, distributed computing, GPU computing

ACM Reference Format:
Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356223

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356223

1 INTRODUCTION

The objective of SLATE is to provide fundamental dense linear algebra capabilities for the high-performance computing (HPC) community at large. The ultimate objective of SLATE is to replace ScaLAPACK [7], which has become the industry standard for dense linear algebra operations in distributed memory environments. 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. In the past two decades, HPC has witnessed tectonic shifts in hardware technology, software technology, and a plethora of algorithmic innovations in scientific computing that are not reflected in ScaLAPACK. At the same time, while several libraries provide functionality similar to ScaLAPACK, no replacement for ScaLAPACK has emerged to gain widespread adoption in applications. SLATE aims to be this replacement, boasting superior performance and scalability in modern, heterogeneous, distributed-memory HPC environments.

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).

SLATE currently implements parallel BLAS (PBLAS); linear systems using partial pivoted LU, Cholesky, and block Aasen's for symmetric indefinite systems [6]; mixed-precision iterative refinement; least squares using communication-avoiding QR [15]; and matrix inversion. We are actively developing communication-avoiding two-stage symmetric eigenvalue and singular value decompositions. Future work will include non-symmetric eigenvalue problems and randomized algorithms.

1.1 Motivation

In the United States, the plan for achieving 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.

The urgency of developing multi-GPU accelerated, distributed-memory software is underscored by the architectures of these systems. The Summit system contains three NVIDIA V100 GPUs per POWER9 CPU. The peak double precision floating point performance of the CPUs is 22 cores $\times$ 24.56 GFLOP/s = 0.54 TFLOP/s. The peak performance of the GPUs is 3 devices $\times$ 7.8 TFLOP/s = 23.4 TFLOP/s. That is, 97.7% of performance is on the GPU side, and only 2.3% of performance is on the CPU side. This means that, for all practical purposes, the CPUs can be ignored as floating point hardware, with their main purpose being to keep the GPUs busy.

1.2 Related work

Numerous libraries provide dense linear algebra routines. Since its initial release nearly 30 years ago, LAPACK [5] has become the de facto standard library for dense linear algebra on a single node. It leverages vendor-optimized BLAS for node-level performance, including shared-memory parallelism. ScaLAPACK [7] built upon LAPACK by extending its routines to distributed computing, relying on both the Parallel BLAS (PBLAS) and explicit distributed-memory parallelism. Some attempts [18, 19] have been made to adapt the ScaLAPACK library for accelerators, but these efforts have shown the need for a new framework.

More recently, the DPLASMA [9] and Chameleon [2, 3] libraries both build a task dependency graph and launch tasks as their dependencies are fulfilled. This eliminates the artificial synchronizations inherent in ScaLAPACK's design, and allows for overlap of communication and computation. DPLASMA relies on the PaRSEC runtime to schedule tasks, while Chameleon uses the StarPU runtime. If the dataflow is known at compile time, an algorithm can be expressed using the efficient and scalable Parameterized Task Graph (PTG) interface. Algorithms that make runtime decisions based on data, such as partial pivoting in LU, are easier to express using the less efficient but more intuitive Insert Task interface, similar to OpenMP tasks. Both DPLASMA and Chameleon are accelerated on GPU-based architectures. Instead of leveraging C++ templates, they auto-generate all precisions from the double-complex code.

Elemental [39] is an extensive distributed linear algebra library, using C++ templates for multiple precision support. It is fairly unique in distributing the matrix by elements, rather than blocks, thus allowing arbitrary algorithmic blocking instead of tying the blocking to the distribution. However, it lacks accelerator support.

Algorithmic innovations to minimize communication and increase computational intensity have also been explored. This includes the two-stage reductions to tridiagonal and bidiagonal for the symmetric eigenvalue and SVD problems, respectively. ELPA [37] provides symmetric eigensolvers, with both one stage and two stage reductions to tridiagonal, and has GPU acceleration, but lacks other ScaLAPACK functionality. Demmel et al. have also derived communication-avoiding algorithms for QR and LU factorization [15], as well as for matrix-matrix multiplication [14, 42].

There are also numerous node-level linear algebra libraries. Eigen [32] and Armadillo [41] use C++ templates extensively to make precision-independent code. MAGMA [16] provides accelerated linear algebra, with extensive coverage of LAPACK's functionality. ViennaCL [40] provides accelerated linear algebra, but lacks most dense linear algebra aside from BLAS and no-pivoting LU. NVIDIA cuBLAS [38] and AMD rocBLAS [4] provide native BLAS operations on GPU accelerators. Vendor libraries such as Cray Scientific Libraries (LibSci) [13] and IBM Engineering and Scientific Subroutine Library (ESSL) [29] provide accelerated versions of BLAS and LAPACK for GPU accelerators. These accelerated node-level libraries may give some limited benefit to ScaLAPACK, which is built on top of BLAS and LAPACK. However, this is limited by the bulk-synchronous, fork-join design of ScaLAPACK. The Intel Math Kernels Library (MKL) [31] provides BLAS, LAPACK, and ScaLAPACK for CPUs and Xeon Phi accelerators.

1.3 Goals

Ultimately, a distributed linear algebra library built from the ground up to support accelerators, and incorporating state-of-the-art parallel algorithms, will give the best performance. In this manuscript, we describe how we have designed SLATE to achieve this, and describe how it fulfills the following design goals.

- **Targets modern HPC hardware** consisting of a large number of nodes with multicore processors and several hardware accelerators per node.
- **Achieves portable high performance** by relying on vendor optimized standard BLAS, batch BLAS, and LAPACK, and standard parallel programming technologies such as MPI and OpenMP. Using the OpenMP runtime puts less of a burden on applications to integrate SLATE than adopting a proprietary runtime would.
- **Provides scalability** by employing proven techniques in dense linear algebra, such as 2D block cyclic data distribution and communication-avoiding algorithms, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping.
- **Facilitates productivity** by relying on the intuitive *Single Program, Multiple Data* (SPMD) programming model and a set of simple abstractions to represent dense matrices and dense matrix operations.
- **Assures maintainability** by employing useful C++ facilities, such as templates and overloading of functions and operators, with a focus on minimizing code.
- **Eases transition to SLATE** by natively supporting the ScaLAPACK 2D block-cyclic layout and providing a backwards-compatible ScaLAPACK API.

2 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 dense linear algebra software such as ScaLAPACK, Elemental, and DPLASMA, where the local matrix occupies a contiguous memory region on each process. While

[1] https://www.olcf.ornl.gov/summit/
[2] https://hpc.llnl.gov/hardware/platforms/sierra

DPLASMA uses tiled algorithms, the tiles are stored in one contiguous memory block on each process. In contrast, SLATE makes tiles first class objects 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, easing an application's transition from ScaLAPACK to SLATE. A similar strategy of allocating tiles individually has been successfully used in low-rank, data-sparse linear algebra libraries, such as hierarchical matrices [8, 25] in HLib [26] and with the block low-rank (BLR) format [45]. Compared to other distributed dense linear algebra formats, SLATE's matrix structure offers numerous advantages:

First, the same structure can be used for holding many different matrix types: general, symmetric, triangular, band, symmetric band, etc., as shown in Fig. 1. Little memory is wasted for storing parts of the matrix that hold no useful data, e.g., the upper triangle of a lower triangular matrix. Instead of wasting $O(n^2)$ memory as ScaLAPACK does, only $O(nn_b)$ memory is wasted in the diagonal tiles for a block size n_b; all unused off-diagonal tiles are simply never allocated. There is no need for using complex matrix layouts such as the *Recursive Packed Format* (RPF) [22] or *Rectangular Full Packed* (RFP) [24] in order to save space.

Second, the matrix can be easily converted, in parallel, from one layout to another with $O(P)$ memory overhead for P processors (cores/threads). Possible conversions include: changing tile layout from column-major to row-major, "packing" of tiles for efficient BLAS execution [30], and low-rank compression of tiles. 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 [23].

Also, tiles can be easily allocated and copied among different memory spaces. Both inter-node communication and intra-node communication is vastly simplified. Tiles can be easily and efficiently transferred between nodes using MPI. Tiles can be easily moved in and out of fast memory, such as the MCDRAM in Xeon Phi processors. Tiles can also be copied to one or more device memories in the case of GPU acceleration.

In practical terms, a SLATE matrix is implemented using the `std::map` container from the C++ standard library as:

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

The map's key is a triplet consisting of the tile's (i, j) block row and column indices in the matrix, and the device number (host or accelerator) where the tile is located. It relies on global indexing of tiles, meaning that each tile is identified by the same unique tuple across all processes. The map's value is a lightweight Tile object that stores a tile's data and 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. Each node stores only its local subset of tiles, as shown in Fig. 2. Mapping of tiles to nodes is defined by a C++ lambda function, and set to 2D block cyclic mapping by default, but the user can supply an arbitrary mapping function. Similarly, distribution to accelerators within each node is 1D block cyclic by default, but the user can substitute an arbitrary function.

Remote access is realized by replicating remote tiles in the local matrix for the duration of the operation. This is shown in Fig. 2 for the trailing matrix update in Cholesky, where portions of the remote panel (yellow) have been copied locally.

Finally, SLATE can support non-uniform tile sizes (Fig. 3). Most factorizations require that the diagonal tiles are square, but the block row heights and block column widths can, in principle, be arbitrary. The current implementation has a fixed tile size, but the structure does not require it. This will facilitate applications where the block structure is significant, for instance in *Adaptive Cross Approximation* (ACA) linear solvers [34].

3 MATRIX CLASS HIERARCHY

The design of SLATE revolves around the Tile class and the Matrix class hierarchy listed below. 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.

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 known implicitly 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 known implicitly by symmetry ($a_{j,i} = \overline{a_{i,j}}$).

The `BaseMatrix` class stores the matrix dimensions; whether the matrix is upper, lower, or general; whether it is non-transposed, transposed, or conjugate-transposed; how the matrix is distributed; and the set of tiles – both local tiles and temporary workspace tiles as needed during the computation. It also stores the distribution parameters and MPI communicators that would traditionally be stored in a ScaLAPACK context. As such, there is no separate structure to maintain state, nor any need to initialize or finalize the SLATE library.

It is also planned to add a similar hierarchy for band matrices: `BandMatrix`, `TriangularBandMatrix`, `SymmetricBandMatrix`, and `HermitianBandMatrix` classes. For an upper block bandwidth k_u and lower block bandwidth k_l, only the tiles $A(i, j)$ for $j - k_u \leq i \leq j + k_l$ are stored.

SLATE routines require the correct matrix types for their arguments, which helps to ensure correctness, while inexpensive shallow copy conversions exist between the various matrix types. For instance, a general `Matrix` can be converted to a `TriangularMatrix`

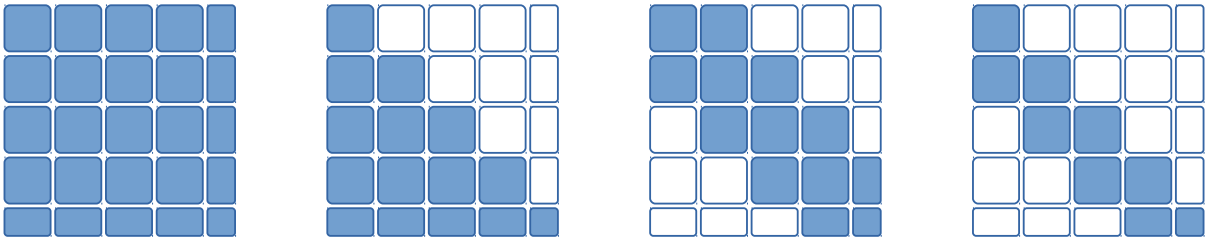

Figure 1: General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored.

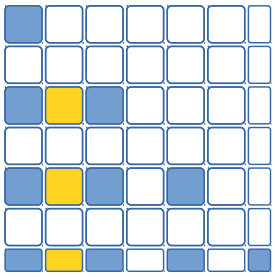

Figure 2: View of symmetric matrix on process (0, 0) in 2×2 process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1).

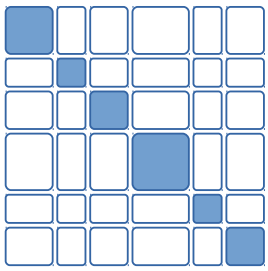

Figure 3: Block sizes can vary. Most algorithms require square diagonal tiles.

for doing a triangular solve (`trsm`), without copying. The two matrices have a reference-counted C++ shared pointer to the same underlying data (`std::map` of tiles).

Likewise, copying a matrix object is an inexpensive shallow copy, using a C++ shared pointer. Sub-matrices 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.

Transpose and conjugate-transpose are supported by creating an inexpensive shallow copy and changing the transposition operation flag stored in the new matrix object. For a matrix `A` that is a possibly transposed copy of an original matrix `A0`, the function `A.op()` returns `Op::NoTrans`, `Op::Trans`, or `Op::ConjTrans`, indicating whether `A` is non-transposed, transposed, or conjugate-transposed, respectively. The functions `A = transpose(A0)` and `A = conj_transpose(A0)` return new matrices with the operation flag set appropriately. Querying properties of a matrix object takes the transposition and sub-matrix offsets into account. For instance, `A.mt()` is the number of block rows of $\text{op}(A_0)$, where $A = \text{op}(A_0) = A_0$, A_0^T, or A_0^H. The function `A(i, j)` returns the i, j-th tile of $\text{op}(A_0)$, with the tile's operation flag set to match the `A` matrix.

SLATE supports upper and lower storage with `A.uplo()` returning `Uplo::Upper` or `Uplo::Lower`. Tiles likewise have a flag indicating upper or lower storage, accessed by `A(i, j).uplo()`. For tiles on the matrix diagonal, the `uplo` flag is set to match the matrix, while for off-diagonal tiles it is set to `Uplo::General`.

3.1 Handling of side, uplo, trans

The classical BLAS take 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 the PLASMA [17] 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, at the high level, `gemm` can ignore the operations on A and B. If transposed, the matrix object itself handles swapping indices to obtain the correct tiles during the algorithm. At the low level, the transposition operation is set on the tiles, and is passed on to the underlying node-level BLAS `gemm` routine.

Similarly, the Cholesky factorization, shown in Fig. 9, implements only the lower case; the upper case is handled by a shallow copy transposition to map it to the lower case. The data is not physically transposed in memory, only the transpose `op` flag is set so that the matrix is *logically* lower.

4 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. Operations are defined to apply consistently across all precisions. For instance, `blas::conj` extends `std::conj` to apply to real precisions (float, double), where it is a no-op. SLATE's BLAS++ component [21] provides overloaded, precision-independent wrappers for all the underlying node-level BLAS, which SLATE's PBLAS are built on top of. For instance, `blas::gemm` in BLAS++ maps to the classical `sgemm`, `dgemm`, `cgemm`, or `zgemm` BLAS, depending on the precision of its arguments. For real arithmetic, symmetric and Hermitian matrices are considered interchangeable, so `hemm` maps to `symm`, `herk` to `syrk`, and `her2k` to `syr2k`. This mapping aides in templating higher-level routines, such as Cholesky, which does a `herk` (mapped to `syrk` in real) to update the trailing matrix.

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 accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS. For instance, Intel MKL and NVIDIA cuBLAS provide half-precision `gemm` operations.

SLATE also implements mixed-precision algorithms [10] that factor a matrix in low precision, then use iterative refinement to attain a high precision final result. These exploit the faster processing in low precision for the $O(n^3)$ factorization work, while refinement in the slower high precision is only $O(n^2)$ work. In SLATE, the low and high precisions are independently templated; currently we use the traditional single and double combination. However, recent interest with half precision has led to algorithms using half precision with either single or double [11, 27]. One could also go to higher precisions, using double-double [28] or quad for the high precision. By adding the relevant underlying node-level BLAS operations in the desired precisions to BLAS++, the templated nature of SLATE greatly simplifies instantiating different combinations of precisions.

5 PARALLELISM MODEL

SLATE utilizes three or four levels of parallelism: distributed parallelism between nodes using MPI, explicit thread parallelism using OpenMP, implicit thread parallelism within the vendor's node-level BLAS, and, at the lowest level, vector parallelism for the processor's SIMD vector instructions. For multicore, SLATE typically uses all the threads explicitly, and uses the vendor's BLAS in sequential mode. For GPU accelerators, SLATE uses a batch BLAS call, utilizing the thread-block parallelism built into the accelerator's BLAS.

The cornerstones of SLATE are 1) the SPMD programming model for productivity and maintainability, 2) dynamic task scheduling using OpenMP for maximum node-level parallelism and portability, 3) the *lookahead* technique for prioritizing the *critical path*, 4) primarily reliance on the 2D block cyclic distribution for scalability, 5) reliance on the gemm operation, specifically its batch rendition, for maximum hardware utilization.

The Cholesky factorization demonstrates the basic framework, with its task graph shown in Fig. 4 and code shown in Fig. 9. Dataflow tasking (`omp task depend`, Fig. 9 lines 19, 48, 68) is used for scheduling operations with dependencies on large blocks

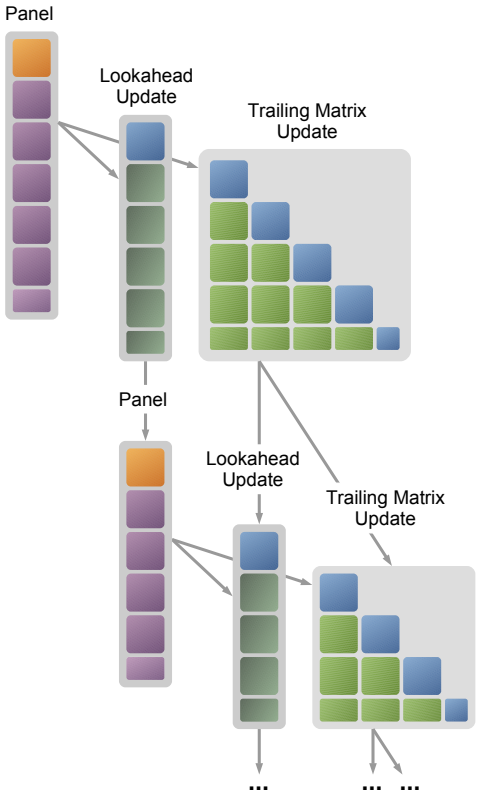

Figure 4: Tasks in Cholesky factorization. Arrows depict dependencies.

of the matrix. Dependencies are performed on a dummy vector, representing each block column in the factorization, rather than on the matrix data itself. Within each large block, either nested tasking (`omp task`, Fig. 10 line 12) or batch operations of independent tile operations are used for scheduling individual tile operations to individual cores, without dependencies. For accelerators, batch BLAS calls are used for fast processing of large blocks of the matrix using accelerators.

Compared to pure tile-by-tile dataflow scheduling, as used by DPLASMA and Chameleon, this approach minimizes the size of the task graph and number of dependencies to track. For a matrix of $N \times N$ tiles, tile-by-tile scheduling creates $O(N^3)$ tasks and dependencies, which can lead to significant scheduling overheads. This is one of the main performance handicaps of the OpenMP version of the PLASMA library [17] in the case of manycore processors such as the Xeon Phi family. In contrast, the SLATE approach creates $O(N)$ dependencies, eliminating the issue of scheduling overheads. At the same time, this approach is a necessity for scheduling a large set of independent tasks to accelerators, to fully occupy their massive compute resources. It also eliminates the need to use a hierarchical task graph to satisfy the vastly different levels of parallelism on CPUs vs. on accelerators [47].

At each step of Cholesky, one or more columns of the trailing submatrix are prioritized for processing, using the OpenMP `priority` clause, to facilitate faster advance along the critical path,

implementing a lookahead. At the same time, the lookahead depth needs to be limited, as it is proportional to the amount of extra memory required for storing temporary tiles. Deep lookahead translates to depth-first processing of the task graph, synonymous with left-looking algorithms, but can also lead to catastrophic memory overheads in distributed memory environments [36].

Distributed memory computing is implemented by filtering operations based on the matrix distribution function (Fig. 10 line 11); in most cases, the owner of the output tile performs the computation to update the tile. Appropriate communication calls are issued to send tiles to where the computation will occur. Management of multiple accelerators is handled by a node-level memory consistency protocol.

The user can choose among various target implementations. In the case of accelerated execution, the updates are executed as calls to batch gemm (`Target::Devices`). In the case of multicore execution, the updates can be executed as:

- a set of OpenMP tasks (`Target::HostTask`),
- a nested parallel for loop (`Target::HostNest`), or
- a call to batch gemm (`Target::HostBatch`).

To motivate our choices of CPU tasks on individual tiles and GPU tasks using batches of tiles, we examine the performance of `dgemm`. Libraries such as DPLASMA and Chameleon have demonstrated that doing operations on a tile-by-tile basis can achieve excellent CPU performance. For instance, as shown in Fig. 5, for tile sizes ≥ 160, IBM ESSL dgemm achieves over 90% of its maximum performance. In contrast, accelerators would take much larger tiles to reach their maximum performance. On an NVIDIA P100, cuBLAS dgemm would require an unreasonably large tile size ≥ 3136 to achieve 90% of its maximum performance. DPLASMA dealt with this disparity in tile sizes between the CPU and GPU by using a hierarchical DAG, whereby the CPU has small tiles and the GPU has large tiles [47].

Instead, in SLATE we observe that most `gemm` operations are block outer-products, where A is a block column and B is a block row (e.g., the Schur complement in LU factorization), and that these can be implemented using a batch gemm. In Fig. 6, the regular cuBLAS dgemm uses standard LAPACK column-major layout, while the tiled / batch dgemm uses a tiled layout with $k \times k$ tiles and multiplies all tiles simultaneously using cuBLAS batch dgemm. This demonstrates that at specific sizes $(192, 256, \ldots)$, the batch dgemm matches the performance of a regular dgemm. Thus with an appropriately chosen, modest block size, SLATE can achieve the maximum performance from accelerators.

SLATE intentionally relies on standards in MPI, OpenMP, and BLAS to maintain easy portability. Any CPU platform with good implementations of these standards should work well for SLATE. For accelerators, SLATE's reliance on batch gemm means any platform that implements batch gemm is a good target. Differences between vendors' BLAS implementations will be abstracted at a low level in the BLAS++ library to ease porting. There are very few accelerator (e.g., CUDA) kernels in SLATE – currently just matrix norms and transposition – so porting should be a lightweight task.

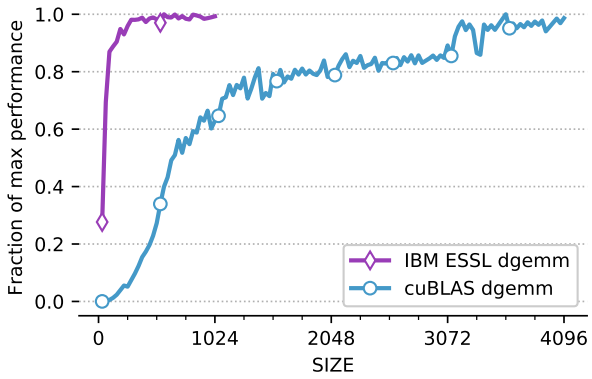

Figure 5: Performance of square `dgemm`, as fraction of maximum single-core ESSL performance (23.6 GFLOP/s) and cuBLAS performance (4560 GFLOP/s), respectively.

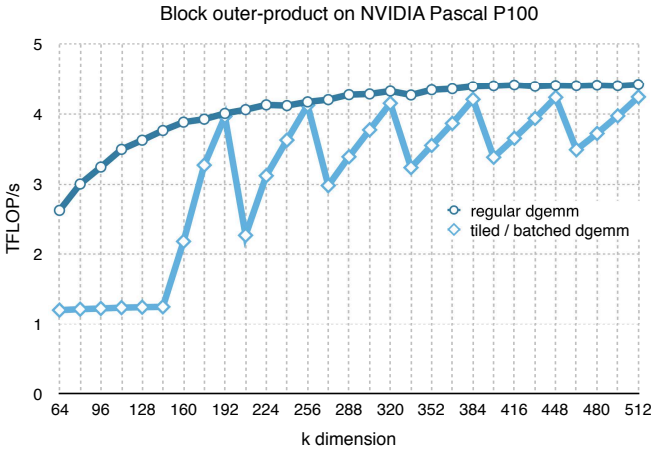

Figure 6: Block outer-product `dgemm`, $C = C - AB$, where C is $40{,}000 \times 40{,}000$, A is $40{,}000 \times k$, B is $k \times 40{,}000$.

6 MESSAGE PASSING COMMUNICATION

Communication in SLATE relies on explicit dataflow information. When a tile will be needed for computation, it is broadcast to all the processes where it is required, as shown in Fig. 7 for broadcasting a single tile from the Cholesky panel to its trailing matrix update. Rather than explicitly listing MPI ranks, the broadcast is expressed in terms of the destination tiles to be updated. `tileBcast` 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 (Fig. 9 lines 25, 40). To optimize communication, `listBcast` aggregates a list of these tile broadcasts and pipelines the MPI and CPU to accelerator communication. 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 tree fashion to broadcast the data.

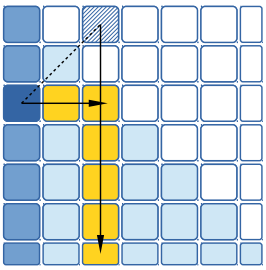

Figure 7: Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix.

7 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.

CUDA managed memory is applied on a memory page basis. However, a tile in a ScaLAPACK-style matrix is not contiguous and has a column stride, meaning each column could be in a separate page, and migrating pages to the GPU would also migrate excess data outside the tile, adding communication overhead. CUDA managed memory is also a proprietary solution and, in particular, the MPI implementation must be CUDA-aware to work with it.

OpenMP directives and offload allow for arbitrary size arrays, not just page sizes, but support only contiguous data – not submatrices with a column stride. This would work if our tiles were always contiguous, but again would prevent us from working on a ScaLAPACK-style layout.

Instead, currently 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 workspace tiles have been deleted.

For offload to GPU accelerators, SLATE implements a memory consistency model, inspired by the MOSI cache coherency protocol [33, 43], 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. SLATE's 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.

```
1  // Distributed parallel Cholesky solve, AX = B
2  // A:    Matrix to factor; overwritten by L
3  // B:    On input, matrix B; overwritten by X
4  // opts: User options such as Target and Lookahead
5  // scalar_t: Datatype: float, double, std::complex, etc.
6  template <typename scalar_t>
7  void posv(HermitianMatrix<scalar_t>& A,
8            Matrix<scalar_t>& B,
9            const std::map<Option, Value>& opts)
10 {
11     potrf( A, opts );     // factor A = LL^H
12     potrs( A, B, opts );  // solve AX = B using factorization
13 }
```

Figure 8: Cholesky solve driver, `slate::posv`

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 (Fig. 10 lines 15–17). The OnHold flag is used to optimize CPU to accelerator communication. Normally, at the end of an operation such as `internal::gemm`, workspace tiles are delete from GPU devices to limit the required workspace memory. However, if 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.

8 SLATE API

Currently, SLATE's routine names are derived from traditional BLAS and LAPACK names, minus the traditional initial letter denoting the precision (`s`, `d`, `c`, `z`). In the future, we will develop simplified names using overloaded functions, using the Matrix types to identify the operation to be performed. For instance, `multiply(A, B, C)` can map to a general, symmetric, or triangular matrix-matrix multiply (`gemm`, `symm`, or `trmm`) depending on whether the types of `A`, `B`, and `C` are general Matrix, SymmetricMatrix, or TriangularMatrix, respectively.

SLATE's API is composed of several layers, going from highest to lowest level:

Drivers. As in LAPACK and ScaLAPACK, driver routines solve an entire problem, such as a linear system $Ax = b$ (routines `gesv`, `posv`), a least squares problem $Ax \cong b$ (`gels`), or a singular value decomposition $A = U\Sigma V^H$ (`gesvd`). Drivers in turn call computational routines to solve sub-problems. Drivers are typically independent of the target (CPU or device), delegating those details to lower level routines. Figure 8 gives an example of the Cholesky driver, `posv`, which relies on computational routines `potrf` and `potrs` to factor the matrix A and solve the system $Ax = b$.

Computational routines. Again as in LAPACK and ScaLAPACK, computational routines solve a sub-problem, such as computing an LU factorization (`getrf`), or solving a linear system given an LU factorization (`getrs`). In SLATE, these are templated on target (CPU or device), with the code typically independent of the device.

However, if needed, code can be optimized for a specific target by providing an overloaded version. Communication between processes and dependencies between tasks are managed at this level. SLATE's PBLAS exists at this level.

Figure 9 gives an example of the Cholesky factorization computational routine (`potrf`), used by the Cholesky driver. SLATE's `potrf` routine is approximately the same length as the LAPACK `dpotrf` code, and roughly half the length of the ScaLAPACK and MAGMA code (all excluding comments). Yet SLATE's code handles all precisions, multiple targets, distributed memory and shared memory parallelism, a lookahead to overlap communication and computation, and GPU acceleration. Of course, there is significant code in lower levels, but this demonstrates that writing driver and computational routines can be simplified by delegating code complexity to lower level abstractions.

Internal routines. SLATE adds a third layer of internal routines that generally perform one step of a computational routine. For instance, in the outer k loop, `slate::gemm` calls a sequence of `slate::internal::gemm`, each of which performs one block outer product. Most internal routines consist of a set of independent tile operations that can be issued as a batch gemm or an OpenMP parallel-for loop, with no task dependencies to track. Internal routines provide device-specific implementations such as OpenMP nested tasks, parallel-for loops, or batch BLAS operations. In many linear algebra algorithms, these internal routines implement the trailing matrix update.

Figure 10 gives an example of the internal gemm routine, used in the PBLAS gemm routine and for the update in the Cholesky factorization routine. This code reveals several features of SLATE. Currently, routines loop over all tiles in the matrix C, and selects just the local tiles to operate on. By filtering for local tiles via the `tileIsLocal` call, SLATE is agnostic to the actual distribution. To reduce overheads, we are developing 2D iterators that are aware of the distribution, so can iterate over just the local tiles without needing to check if tiles are local, while the code can still be agnostic to the distribution.

In the `potrf` call, the `internal::gemm` call is an OpenMP task. Within `internal::gemm`, each tile gemm call is a nested OpenMP task, with no dependencies. Before each tile gemm, `tileGetForReading` and `tileGetForWriting` ensures that the tiles are in CPU memory, initiating a transfer from accelerator memory if necessary. Remote tiles are given a life counter to track the number of tiles they update. After each tile gemm, the A and B tiles have their lives decremented by `tileTick`; once all local tiles in row i of C are updated, the life of tile $A(i, 0)$ reaches zero and the tile is deleted if it is a workspace tile (i.e., not an origin tile). Similarly, when all local tiles in column j of C are updated, the life of tile $B(0, j)$ reaches zero and the tile is deleted, if it is workspace.

Panel operations, such as the LU and QR parallel panels, also exist as internal routines. However, unlike trailing matrix updates, panels create a set of interdependent tasks.

Tile operations. These update one or a small number of individual tiles. For instance, a tile gemm takes three tiles, A, B, and C, and updates C. Transposition of individual tiles is resolved at this level when calling optimized BLAS. This allows higher level operations to ignore whether a matrix is transposed or not. Currently, all tile operations are CPU-only, since accelerators use only batch operations. Figure 11 gives an example of the tile gemm routine, used in the internal gemm routine (Fig. 10).

BLAS++ and LAPACK++. At the lowest level, these packages provide thin, precision independent, overloaded C++ wrappers around tradition BLAS, batch BLAS, and LAPACK routines, as previously discussed in Section 4. They use C++ calling conventions and enum values instead of character constants, but otherwise the calling sequence is similar to the standard BLAS and LAPACK routines. BLAS++ also includes batch BLAS, on both CPUs and GPUs.

A slightly higher level interface taking arrays as `mdspan` objects may be developed as `mdspan` becomes standardized [20] and wide-spread in C++ standard library implementations. That would eliminate the separate dimension arguments, yielding, for instance,

```
gemm( transA, transB, alpha, A, B, beta, C )
```

where `A`, `B`, and `C` are `mdspan` objects encapsulating their dimensions and column or row strides.

9 SCALAPACK COMPATIBILITY

As many applications have significant existing code bases using ScaLAPACK, we also 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 $n_b = 64$ used for ScaLAPACK, SLATE may require $n_b = 192$ or 256 for good efficiency. Using SLATE's native API directly also avoids overhead in creating the SLATE matrices inside each intercepted ScaLAPACK call. Thus, in the long term, we recommend rewriting applications to natively call SLATE. In the future, we plan to provide thin C and Fortran wrappers to access SLATE's functionality directly.

10 RESULTS

In this paper we have focused on the SLATE framework itself, rather than the specific linear algebra algorithms that SLATE implements. We present performance for a couple routines here, to demonstrate SLATE's capabilities. More detailed analysis and performance results will be provided in forthcoming reports and papers that focus on specific algorithms.

Performance numbers were collected using the SummitDev system [3] at ORNL, which is a prototype one generation behind ORNL's flagship supercomputer, Summit. (These results were obtained before Summit entered production.) SummitDev contains 54 nodes, each of which contain two IBM POWER8 CPUs, ten cores each, with 256 GiB of DDR4 memory, and four NVIDIA P100 (Pascal) accelerators with 16 GiB of HBM2 memory each. The accelerators

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

```
// Distributed parallel Cholesky factorization, A = L L^H
template <Target target, typename scalar_t>
void potrf(slate::internal::TargetType<target>,
           HermitianMatrix<scalar_t> A, int64_t look)
{
    using real_t = blas::real_type<scalar_t>;
    scalar_t one = 1.0;
    const int64_t A_nt = A.nt();
    if (A.uplo() == Uplo::Upper) // if upper, change to lower
        A = conj_transpose(A);
    // dummy vector to track dependencies
    std::vector<uint8_t> column_vector(A_nt);
    uint8_t* column = column_vector.data();

    #pragma omp parallel
    #pragma omp master
    for (int64_t k = 0; k < A_nt; ++k) {
        // panel, high priority
        #pragma omp task depend(inout:column[k]) priority(1)
        {
            // factor A(k, k)
            internal::potrf<Target::HostTask>(A.sub(k, k), 1);
            // send A(k, k) down col A(k+1:nt-1, k)
            if (k+1 <= A_nt-1)
                A.tileBcast(k, k, A.sub(k+1, A_nt-1, k, k));
            // A(k+1:nt-1, k) * A(k, k)^{-H}
            if (k+1 <= A_nt-1) {
                auto Akk = A.sub(k, k);
                auto Tkk = TriangularMatrix<scalar_t>(
                    Diag::NonUnit, Akk);
                internal::trsm<Target::HostTask>(
                    Side::Right,
                    one, conj_transpose(Tkk),
                    A.sub(k+1, A_nt-1, k, k), 1);
            }
            typename Matrix<scalar_t>::BcastList bcast_list_A;
            for (int64_t i = k+1; i < A_nt; ++i) {
                // send A(i, k) across row A(i, k+1:i)
                //           and down col A(i:nt-1, i)
                bcast_list_A.push_back(
                    {i, k, {A.sub(i, i, k+1, i),
                            A.sub(i, A_nt-1, i, i)}});
            }
            A.template listBcast(bcast_list_A);
        } // omp task

        // update lookahead column(s), high priority
        for (int64_t j = k+1; j < k+1+look && j < A_nt; ++j) {
            #pragma omp task depend(in:column[k]) \
                             depend(inout:column[j]) priority(1)
            {
                // A(j, j) -= A(j, k) * A(j, k)^H
                internal::herk<Target::HostTask>(
                    real_t(-1.0), A.sub(j, j, k, k),
                    real_t( 1.0), A.sub(j, j), 1);

                // A(j+1:nt-1, j) -= A(j+1:nt-1, k) * A(j, k)^H
                if (j+1 <= A_nt-1) {
                    auto Ajk = A.sub(j, j, k, k);
                    internal::gemm<Target::HostTask>(
                        -one, A.sub(j+1, A_nt-1, k, k),
                              conj_transpose(Ajk),
                         one, A.sub(j+1, A_nt-1, j, j), 1);
                }
            }
        }
        // update trailing submatrix, normal priority
        if (k+1+look < A_nt) {
            #pragma omp task depend(in:column[k]) \
                             depend(inout:column[k+1+look]) \
                             depend(inout:column[A_nt-1])
            {
                // A(kl+1:nt-1, kl+1:nt-1) -=
                //     A(kl+1:nt-1, k) * A(kl+1:nt-1, k)^H
                // where kl = k + look
                internal::herk<target>(
                    real_t(-1.0), A.sub(k+1+look, A_nt-1, k, k),
                    real_t( 1.0), A.sub(k+1+look, A_nt-1));
            }
        }
    } // k loop

    A.tileUpdateAllOrigin();
    A.releaseWorkspace();
}
```

Figure 9: Cholesky factorization computational routine, `slate::potrf`, with lookahead depth *look*.

are connected by NVLink 1.0 at 80 GB/s. The nodes are connected with a fat-tree enhanced data rate (EDR) InfiniBand network.

All runs were performed using 16 nodes, yielding a peak multicore performance of 8.96 TFLOP/s and a peak accelerator performance of 339.2 TFLOP/s in double precision. 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.

The software environment used for the experiments included GNU gcc 7.1.0, CUDA 9.0.69, ESSL 5.5.0, Spectrum MPI 10.1.0.4, Netlib LAPACK 3.6.1, and Netlib ScaLAPACK 2.0.2.

10.1 Parallel BLAS

SLATE's PBLAS [35] implements all Level 3 BLAS operations. For the basic matrix-matrix multiply (gemm) operation, $C = \alpha AB + \beta C$, we follow the SUMMA algorithm [1, 44], also used in ScaLAPACK's PBLAS [12]. At each step k, tiles in block column k of A are broadcast horizontally across the process grid, while tiles in block row k of B are broadcast vertically across the process grid. Then all tiles of C are updated in parallel by the process that owns them, according to $C_{ij} = \alpha A_{ik} B_{kj} + C_{ij}$. Communication is pipelined, so with *lookahead* = 1, the broadcast of step $k+1$ occurs during the computation of step k, to overlap communication and computation. As previously mentioned in Section 3.1, PBLAS operations can ignore the transposition operations, which are resolved at lower levels.

When C is small relative to A and B, this algorithm would produce load imbalance and an excessive amount of communication, so alternate algorithms are used.

Performance for SLATE's gemm in double precision is given in Fig. 12. In the left graph, we see that using only CPUs, SLATE performs comparably with ScaLAPACK, except for small matrices. Both achieve a substantial portion of the 8.96 TFLOP/s theoretical peak. SLATE's results used a relatively large block size $n_b = 512$; tuning n_b would improve performance for small matrix sizes. We also expect optimizations such as launching multiple asynchronous MPI_Isend will improve network performance. For the accelerated performance curve in the right graph, the performance massively outperforms the multicore performance, but the curve is basically

```
// C = alpha AB + beta C; A is one block col, B is one block row
template <typename scalar_t>
void gemm(internal::TargetType<Target::HostTask>,
          scalar_t alpha, Matrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          scalar_t beta,  Matrix<scalar_t>& C,
          int priority, Layout layout)
{
    for (int64_t i = 0; i < C.mt(); ++i) {
        for (int64_t j = 0; j < C.nt(); ++j) {
            if (C.tileIsLocal(i, j)) {
                #pragma omp task shared(A, B, C, err) \
                                 priority(priority)
                {
                    A.tileGetForReading(i, 0);
                    B.tileGetForReading(0, j);
                    C.tileGetForWriting(i, j);
                    gemm(alpha, A(i, 0),
                                B(0, j),
                         beta,  C(i, j));
                    A.tileTick(i, 0);
                    B.tileTick(0, j);
                }
            }
        }
    }

    #pragma omp taskwait
}
```

Figure 10: Internal matrix multiply routine, `slate::internal::gemm`, corresponding to a single block outer product.

```
// C = alpha AB + beta C
template <typename scalar_t>
void gemm(
    scalar_t alpha, Tile<scalar_t> const& A,
                    Tile<scalar_t> const& B,
    scalar_t beta,  Tile<scalar_t>& C)
{
    if (C.op() == Op::NoTrans) {
        // C = opA(A) opB(B) + C
        blas::gemm(blas::Layout::ColMajor,
                   A.op(), B.op(),          // transpositions
                   C.mb(), C.nb(), A.nb(),  // tile dimensions
                   alpha, A.data(), A.stride(),
                          B.data(), B.stride(),
                   beta,  C.data(), C.stride());
    }
    else { ... }
}
```

Figure 11: Tile matrix multiply routine, `slate::gemm`. Cases for transposed C (C^T and C^H) are omitted.

linear. The reason for this behavior is that the PBLAS performance profile follows the roofline model [46]. For the tested matrix sizes, the performance is bound by communication and therefore not near saturation. This is unsurprising given the ratio of Summit-Dev's node performance to its communication bandwidth – today's machines are, unfortunately, over-provisioned for floating point operations and under-provisioned for network bandwidth.

Performance for the standard four precisions (single, double, single-complex, double-complex) is presented in Fig. 13 and Table 1. For multicore runs (Fig. 13, left), the performance numbers are as expected. Single precision is about twice as fast as double precision, which comes directly from the 2× factor between the single-precision peak and the double-precision peak of the POWER8 cores. At the same time, complex arithmetic is slightly faster than real arithmetic. This is because complex arithmetic is twice as compute-intensive as real arithmetic, bringing the performance number a notch closer to the hardware peak.

precision	max CPU performance	max GPU performance
double	7.8 TFLOP/s	107.8 TFLOP/s
double-complex	8.0 TFLOP/s	246.8 TFLOP/s
single	14.6 TFLOP/s	166.5 TFLOP/s
single-complex	15.6 TFLOP/s	428.7 TFLOP/s

Table 1: Maximum `gemm` performance attained in Fig. 13. Double precision `gemm` achieves 170.4 TFLOP/s in Fig. 12 due to large sizes tested.

For accelerated runs (Fig. 13, right), the situation is very different. Here, the performance of single precision is twice the performance of double precision; but also, the performance of complex arithmetic is twice the performance of real arithmetic. This is because here the performance is bound not by the floating-point peak, but by the communication bandwidth. As a result, the accelerated performance is directly correlated with the ratio of computation to communication, which is 2× higher in single precision than in double precision, and 2× higher in complex arithmetic than in real arithmetic.

10.2 Linear solvers

For solving a linear system, $AX = B$, SLATE provides the traditional partial pivoting LU factorization for general matrices (`gesv`) and Cholesky factorization for symmetric positive definite matrices (`posv`). The Cholesky factorization is straight-forward to implement using tasks, as previously shown in Fig. 4 and Fig. 9. The LU factorization uses a distributed and multi-threaded parallel panel. This is a challenge with the current OpenMP task environment, as it requires having multiple tasks that are executed together, or equivalently, having a single task with nested parallelism. Currently, we over-subscribe threads to avoid deadlock, but are looking for enhancements to OpenMP to remedy the issue. For least squares problems, SLATE implements a communication-avoiding QR algorithm.

Figure 14 shows performance for Cholesky factorization in double precision. For multicore runs (left), SLATE asymptotically slightly outperforms ScaLAPACK, achieving 6.97 TFLOP/s. Both attain a reasonable portion of the theoretical peak performance. For accelerated runs (right), as with the BLAS in Section 10.1, the performance significantly outperforms the multicore performance, achieving 65.6 TFLOP/s, but is basically linear due to the roofline model.

11 CONCLUSIONS

Scaling to the full performance on today's heterogeneous HPC machines is very challenging. As we have seen, linear algebra libraries need to take advantage of parallelism at multiple levels (nodes, threads, vectors), manage execution on multiple devices (CPUs

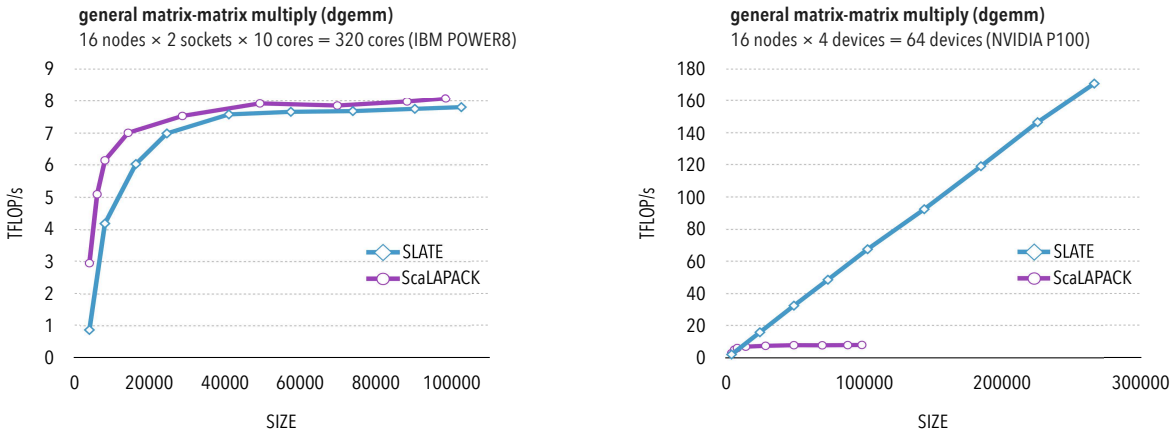

Figure 12: Performance of matrix-matrix multiply (dgemm) without acceleration (left) and with acceleration (right).

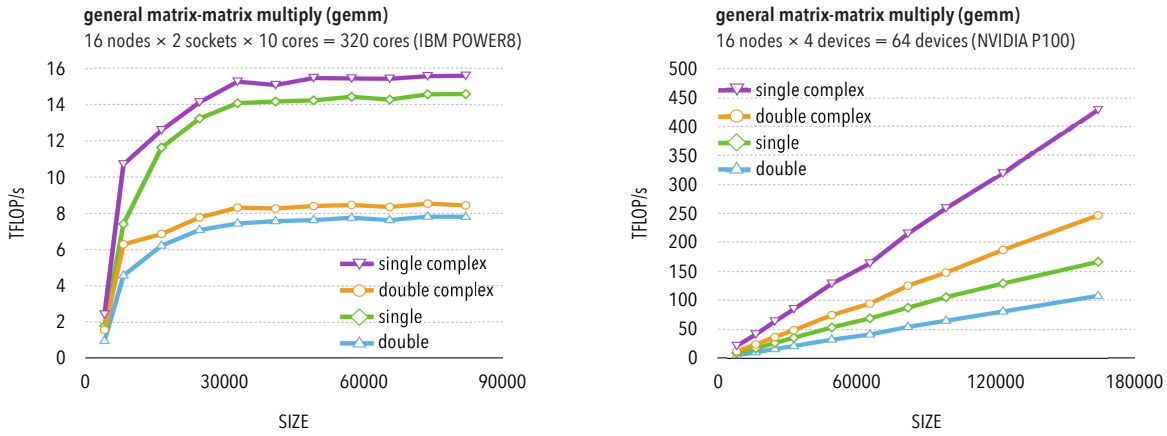

Figure 13: Performance of gemm for different precisions (single/double, real/complex), without acceleration (left) and with acceleration (right).

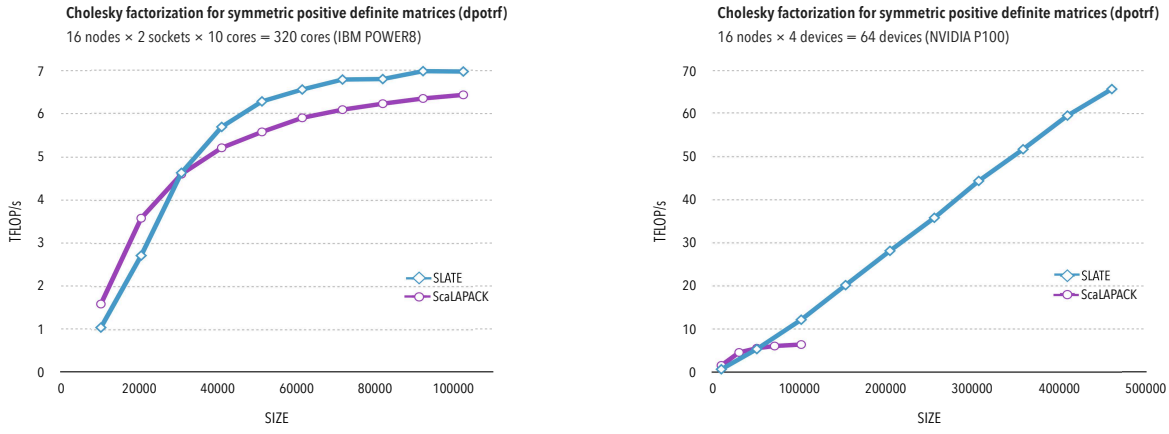

Figure 14: Performance of Cholesky factorization (dpotrf) without acceleration (left) and with acceleration (right).

and accelerators), and minimize communication costs. SLATE addresses these challenges by providing high level abstractions for matrices and tiles, managing parallelism at different levels in large OpenMP tasks with dependencies, and nested parallelism with small independent OpenMP tasks or batch BLAS operations, and using lookahead techniques to overlap communication with computation. A communication-avoiding algorithm is employed for the QR factorization, and similar communication-avoiding algorithms will be used for eigenvalues and the SVD. SLATE provides both a modern, high level C++ API, and a backwards-compatible ScaLAPACK API to aide applications in transition. Initial results show promising performance using multiple nodes and multiple accelerators per node, but there are still significant opportunities to optimize performance.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] Ramesh C Agarwal, Fred G Gustavson, and Mohammad Zubair. 1994. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development* 38, 6 (1994), 673–681. https://doi.org/10.1147/rd.386.0673

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2012. A hybridization methodology for high-performance linear algebra software for GPUs. In *GPU Computing Gems Jade Edition*. Elsevier, 473–484. https://doi.org/10.1016/B978-0-12-385963-1.00034-4

[3] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. 2017. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems* (2017), 14. https://doi.org/10.1109/TPDS.2017.2766064

[4] AMD Corp. 2018. *rocBLAS*. AMD Corp. https://github.com/ROCmSoftwarePlatform/rocBLAS/wiki

[5] 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, Philadelphia, PA. https://doi.org/10.1137/1.9780898719604

[6] Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. 2014. Communication-avoiding symmetric-indefinite factorization. *SIAM J. Matrix Anal. Appl.* 35, 4 (2014), 1364–1406. https://doi.org/10.1137/130929060

[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, Philadelphia, PA. https://doi.org/10.1137/1.9780898719642

[8] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. 2003. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements* 27, 5 (2003), 405–422. https://doi.org/10.1016/S0955-7997(02)00152-2

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. 2011. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 1432–1441. https://doi.org/10.1109/IPDPS.2011.299

[10] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. 2007. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications* 21, 4 (2007), 457–466. https://doi.org/10.1177%2F1094342007084026

[11] Erin Carson and Nicholas J Higham. 2018. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing* 40, 2 (2018), A817–A847. https://doi.org/10.1137/17M1140819

[12] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R Clinton Whaley. 1995. A proposal for a set of parallel basic linear algebra subprograms. In *International Workshop on Applied Parallel Computing*. Springer, 107–114. https://doi.org/10.1007/3-540-60902-4_13

[13] Cray Inc. 2019. *Cray Scientific and Math Libraries (CSML)*. Cray Inc. https://pubs.cray.com/content/S-2529/17.05/xctm-series-programming-environment-user-guide-1705-s-2529/cray-scientific-and-math-libraries-csml

[14] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 261–272. https://doi.org/10.1109/IPDPS.2013.80

[15] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239. https://doi.org/10.1137/080731992

[16] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical computations with GPUs*. Springer, 3–28. https://doi.org/10.1007/978-3-319-06548-9_1

[17] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šíšístek, David Stevens, Mawussi Zounon, and Samuel d. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Transactions on Mathematical Software (TOMS)* 45 (2019), 16:1–16:35. Issue 2. https://doi.org/10.1145/3264491

[18] Peng Du, Stanimire Tomov, and Jack Dongarra. 2012. *Providing GPU capability to LU and QR within the ScaLAPACK framework*. Technical Report UT-CS-12-699, LAPACK Working Note 272. Innovative Computing Laboratory, University of Tennessee. http://www.netlib.org/lapack/lawnspdf/lawn272.pdf

[19] E D'Azevedo and Judith C Hill. 2012. Parallel LU factorization on GPU cluster. *Procedia Computer Science* 9 (2012), 67–75. https://doi.org/10.1016/j.procs.2012.04.008

[20] H. Carter Edwards, Bryce Adelstein Lelbach, Daniel Sunderland, David Hollman, Christian Trott, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, and Daniel Sunderland. 2018. *P0009r7 : mdspan: A Non-Owning Multidimensional Array Reference*. ISO. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0009r7.html

[21] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. 2017. *C++ API for BLAS and LAPACK*. Technical Report ICL-UT-17-03, SLATE Working Note 2. Innovative Computing Laboratory, University of Tennessee. https://www.icl.utk.edu/publications/swan-002

[22] 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* 1541 (1998), 195–206. https://doi.org/10.1007/BFb0095337

[23] 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. https://doi.org/10.1145/2168773.2168775

[24] Fred G Gustavson, Jerzy Waśniewski, Jack J Dongarra, and Julien Langou. 2010. Rectangular full packed format for Cholesky's algorithm: factorization, solution, and inversion. *ACM Transactions on Mathematical Software (TOMS)* 37, 2 (2010), 18. https://doi.org/10.1145/1731022.1731028

[25] Wolfgang Hackbusch. 1999. A Sparse Matrix Arithmetic Based on H-Matrices. Part I: Introduction to H-Matrices. *Computing* 62 (1999), 89–108. https://doi.org/10.1007/s006070050015

[26] Wolfgang Hackbusch, Steffen Börm, and Lars Grasedyck. 2012. *HLib 1.4*. Max-Planck-Institut, Leipzig. http://www.hlib.org

[27] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 47. https://doi.org/10.1109/SC.2018.00050

[28] Yozo Hida, Xiaoye S Li, and David H Bailey. 2001. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*. IEEE, 155–162. https://doi.org/10.1109/ARITH.2001.930115

[29] IBM Corp. 2019. *Engineering and Scientific Subroutine Library*. IBM Corp. https://www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html

[30] Intel Corp. 2016. *Introducing the new Packed APIs for GEMM.* Intel Corp. https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm
[31] Intel Corp. 2019. *Intel Math Kernel Library.* Intel Corp. https://software.intel.com/en-us/mkl
[32] Benoît Jacob, Gaël Guennebaud, et al. 2019. *Eigen.* http://eigen.tuxfamily.org
[33] 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. https://www.researchgate.net/profile/Randy_Katz2/publication/234807819_Implementing_a_cache_consistency_protocol/links/0c960518c1b48bb6d6000000.pdf
[34] Stefan Kurz, Oliver Rain, and Sergej Rjasanow. 2002. The adaptive cross-approximation technique for the 3D boundary-element method. *IEEE Transactions on Magnetics* 38, 2 (2002), 421–424. https://doi.org/10.1109/20.996112
[35] Jakub Kurzak, Mark Gates, Asim YarKhan, Ichitaro Yamazaki, Panruo Wu, Piotr Luszczek, Jamie Finney, and Jack Dongarra. 2018. *Parallel BLAS Performance Report.* Technical Report ICL-UT-18-01, SLATE Working Note 5. University of Tennessee. https://www.icl.utk.edu/publications/swan-005
[36] Jakub Kurzak, Piotr Luszczek, Ichitaro Yamazaki, Yves Robert, and Jack Dongarra. 2017. Design and Implementation of the PULSAR Programming System for Large Scale Computing. *Supercomputing Frontiers and Innovations* 4, 1 (2017), 4–26. https://doi.org/10.14529/jsfi170101
[37] Andreas Marek, Volker Blum, Rainer Johanni, Ville Havu, Bruno Lang, Thomas Auckenthaler, Alexander Heinecke, Hans-Joachim Bungartz, and Hermann Lederer. 2014. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter* 26, 21 (2014), 213201. https://iopscience.iop.org/article/10.1088/0953-8984/26/21/213201/meta
[38] NVIDIA Corp. 2019. *cuBLAS.* NVIDIA Corp. https://docs.nvidia.com/cuda/cublas/
[39] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)* 39, 2 (2013), 13. https://doi.org/10.1145/2427023.2427030
[40] Karl Rupp, Florian Rudolf, and Josef Weinbub. 2010. ViennaCL – a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications.* 51–56. http://www.iue.tuwien.ac.at/pdf/ib_2010/Rupp_GPUScA.pdf
[41] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26. https://doi.org/10.21105/joss.00026
[42] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109. https://doi.org/10.1007/978-3-642-23397-5_10
[43] 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. https://doi.org/10.1145/17356.17404
[44] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274. https://doi.org/10.1002/(SICI)1096-9128(199704)9:4%3C255::AID-CPE250%3E3.0.CO;2-2
[45] Clément Weisbecker. 2013. *Improving multifrontal solvers by means of algebraic block low-rank representations.* Ph.D. Dissertation. Institut National Polytechnique de Toulouse-INPT. https://tel.archives-ouvertes.fr/tel-00934939/
[46] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785
[47] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. 2015. Hierarchical DAG scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium.* IEEE, 156–165. https://doi.org/10.1109/IPDPS.2015.56

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran SLATE's gemm and potrf tests on ORNL's SummitDev supercomputer using gcc 7.1.0, CUDA 9.0.69, IBM ESSL 5.5.0, IBM Spectrum MPI 10.1.0.4, Netlib LAPACK 3.6.1, as described in the paper. For comparison, we also ran ScaLAPACK's gemm and potrf tests using Netlib ScaLAPACK 2.0.2.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

`http://doi.org/10.5281/zenodo.3367812`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: SummitDev

Operating systems and versions: Red Hat Enterprise Linux Server release 7.5

Compilers and versions: gcc 7.1.0

Applications and versions: ScaLAPACK 2.0.2

Libraries and versions: CUDA 9.0.69

Key algorithms: gemm, Cholesky

Output from scripts that gathers execution environment information.

```
LMOD_FAMILY_COMPILER_VERSION=16.1.1-beta6
REMOTEHOST=home2.ccs.ornl.gov
MANPATH=/sw/sources/hpss/man:/autofs/nccs-svm1_sw/su
↪  mmitdev/.swci/1-compute/opt/spack/20180914/linux
↪  -rhel7-ppc64le/xl-16.1.1-beta6/spectrum-mpi-10.2.
↪  0.7-20180830-eyo7zxm2piusmyffr3iytmgwdacl67ju/sh
↪  are/man:/sw/summitdev/xl/16.1.1-beta6/xlC/16.1.1
↪  /man/en_US:/sw/summitdev/xl/16.1.1-beta6/xlf/16.
↪  1.1/man/en_US:/sw/summitdev/lmod/7.4.0/rhel7.2_g
↪  nu4.8.5/lmod/lmod/share/man:/opt/ibm/spectrumcom
↪  puting/lsf/10.1/man::
GPU_TARGET=sm30
XALT_ETC_DIR=/sw/summitdev/xalt/1.1.3/etc
XDG_SESSION_ID=6610
_ModuleTable003_=Ny1wcGM2NGxlL3hsLzE2LjEuMS1iZXRhNi9
↪  zcGVjdHJ1bS1tcGkvMTAuMi4wLjctMjAxODA4MzAubHVhIix
↪  bImZ1bGxOYW1lIl09InNwZWN0cnVtLW1waS8xMC4yLjAuNy0
↪  yMDE4MDgzMCIsWyJsb2FkT3JkZXIiXT0yLHByb3BUPXt9LFs
↪  ic3RhdHVzIl09ImFjdGl2ZSIsWyJ1c2VyTmFtZSJdPSJzcGV
↪  jdHJ1bS1tcGkiLH0seGFsdD17WyJmbiJdPSIvc3cvc3VtbWl
↪  0ZGV2L21vZHVsZWZpbGVzL3NpdGUvbGludXgtcmhlbDctcHB
↪  jNjRsZS9Db3JlL3hhbHQvMS4xLjMubHVhIixbImZ1bGxOYW1
↪  lIl09InhhbHQvMS4xLjMiLFsibG9hZE9yZGVyIl09NCxwcm9
↪  wVD17fSxbInN0YXR1cyJdPSJhY3RpdmUiLFsidXNlck5hbWU
↪  iXT0ieGFsdCIsfSx4bD17WyJmbiJdPSIvc3cvc3VtbWl0
HOSTNAME=summitdev-login1
HOST=summitdev-login1
TERM=xterm-256color
SHELL=/bin/tcsh
LMOD_SYSTEM_DEFAULT_MODULES=DefApps
MODULEPATH_ROOT=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu
↪  4.8.5/modulefiles
SSH_CLIENT=160.91.202.153 47540 22
LIBRARY_PATH=/autofs/nccs-svm1_sw/summitdev/.swci/1-
↪  compute/opt/spack/20180914/linux-rhel7-ppc64le/x
↪  l-16.1.1-beta6/spectrum-mpi-10.2.0.7-20180830-ey
↪  o7zxm2piusmyffr3iytmgwdacl67ju/lib
LMOD_PKG=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.5/l
↪  mod/lmod
OLCF_XL_ROOT=/sw/summitdev/xl/16.1.1-beta6
LSF_SERVERDIR=/opt/ibm/spectrumcomputing/lsf/10.1/li
↪  nux3.10-glibc2.17-ppc64le-csm/etc
OMPI_FC=/sw/summitdev/xl/16.1.1-beta6/xlf/16.1.1/bin
↪  /xlf2008_r
LMOD_VERSION=7.4
OLCF_XLSMP_ROOT=/sw/summitdev/xl/16.1.1-beta6/xlsmp/
↪  5.1.1
SSH_TTY=/dev/pts/39
TARG_TITLE_BAR_PAREN=
LSF_LIBDIR=/opt/ibm/spectrumcomputing/lsf/10.1/linux
↪  3.10-glibc2.17-ppc64le-csm/lib
OPAL_PREFIX=/autofs/nccs-svm1_sw/summitdev/.swci/1-c
↪  ompute/opt/spack/20180914/linux-rhel7-ppc64le/xl
↪  -16.1.1-beta6/spectrum-mpi-10.2.0.7-20180830-eyo7
↪  zxm2piusmyffr3iytmgwdacl67ju
GROUP=USER
USER=USER
LMOD_sys=Linux
```

```
LD_LIBRARY_PATH=/autofs/nccs-svm1_sw/summitdev/.swci
↪ /1-compute/opt/spack/20180914/linux-rhel7-ppc64l
↪ e/xl-16.1.1-beta6/spectrum-mpi-10.2.0.7-20180830
↪ -eyo7zxm2piusmyffr3iytmgwdacl67ju/lib:/sw/summitd
↪ ev/xl/16.1.1-beta6/xlsmp/5.1.1/lib:/sw/summitdev
↪ /xl/16.1.1-beta6/xlmass/9.1.1/lib:/sw/summitdev/
↪ xl/16.1.1-beta6/xlC/16.1.1/lib:/sw/summitdev/xl/
↪ 16.1.1-beta6/xlf/16.1.1/lib:/sw/summitdev/xl/16.
↪ 1.1-beta6/lib:/opt/ibm/spectrumcomputing/lsf/10.
↪ 1/linux3.10-glibc2.17-ppc64le-csm/lib:/opt/ibm/s
↪ pectrum_mpi/jsm_pmix/lib
LS_COLORS=rs=0:di=38;5;27:ln=38;5;51:mh=44;38;5;15:p
↪ i=40;38;5;11:so=38;5;13:do=38;5;5:bd=48;5;232;38
↪ ;5;11:cd=48;5;232;38;5;3:or=48;5;232;38;5;9:mi=0
↪ 5;48;5;232;38;5;15:su=48;5;196;38;5;15:sg=48;5;1
↪ 1;38;5;16:ca=48;5;196;38;5;226:tw=48;5;10;38;5;1
↪ 6:ow=48;5;10;38;5;21:st=48;5;21;38;5;15:ex=38;5;
↪ 34:*.tar=38;5;9:*.tgz=38;5;9:*.arc=38;5;9:*.arj=
↪ 38;5;9:*.taz=38;5;9:*.lha=38;5;9:*.lz4=38;5;9:*.
↪ lzh=38;5;9:*.lzma=38;5;9:*.tlz=38;5;9:*.txz=38;5
↪ ;9:*.tzo=38;5;9:*.t7z=38;5;9:*.zip=38;5;9:*.z=38
↪ ;5;9:*.Z=38;5;9:*.dz=38;5;9:*.gz=38;5;9:*.lrz=38
↪ ;5;9:*.lz=38;5;9:*.lzo=38;5;9:*.xz=38;5;9:*.bz2=
↪ 38;5;9:*.bz=38;5;9:*.tbz=38;5;9:*.tbz2=38;5;9:*.
↪ tz=38;5;9:*.deb=38;5;9:*.rpm=38;5;9:*.jar=38;5;9
↪ :*.war=38;5;9:*.ear=38;5;9:*.sar=38;5;9:*.rar=38
↪ ;5;9:*.alz=38;5;9:*.ace=38;5;9:*.zoo=38;5;9:*.cp
↪ io=38;5;9:*.7z=38;5;9:*.rz=38;5;9:*.cab=38;5;9:*
↪ .jpg=38;5;13:*.jpeg=38;5;13:*.gif=38;5;13:*.bmp=
↪ 38;5;13:*.pbm=38;5;13:*.pgm=38;5;13:*.ppm=38;5;1
↪ 3:*.tga=38;5;13:*.xbm=38;5;13:*.xpm=38;5;13:*.ti
↪ f=38;5;13:*.tiff=38;5;13:*.png=38;5;13:*.svg=38;
↪ 5;13:*.svgz=38;5;13:*.mng=38;5;13:*.pcx=38;5;13:
↪ *.mov=38;5;13:*.mpg=38;5;13:*.mpeg=38;5;13:*.m2v
↪ =38;5;13:*.mkv=38;5;13:*.webm=38;5;13:*.ogm=38;5
↪ ;13:*.mp4=38;5;13:*.m4v=38;5;13:*.mp4v=38;5;13:*
↪ .vob=38;5;13:*.qt=38;5;13:*.nuv=38;5;13:*.wmv=38
↪ ;5;13:*.asf=38;5;13:*.rm=38;5;13:*.rmvb=38;5;13:
↪ *.flc=38;5;13:*.avi=38;5;13:*.fli=38;5;13:*.flv=
↪ 38;5;13:*.gl=38;5;13:*.dl=38;5;13:*.xcf=38;5;13:
↪ *.xwd=38;5;13:*.yuv=38;5;13:*.cgm=38;5;13:*.emf=
↪ 38;5;13:*.axv=38;5;13:*.anx=38;5;13:*.ogv=38;5;1
↪ 3:*.ogx=38;5;13:*.aac=38;5;45:*.au=38;5;45:*.fla
↪ c=38;5;45:*.mid=38;5;45:*.midi=38;5;45:*.mka=38;
↪ 5;45:*.mp3=38;5;45:*.mpc=38;5;45:*.ogg=38;5;45:*
↪ .ra=38;5;45:*.wav=38;5;45:*.axa=38;5;45:*.oga=38
↪ ;5;45:*.spx=38;5;45:*.xspf=38;5;45:
LMOD_MPI_NAME=spectrum-mpi
LSF_CONFDIR=/opt/ibm/spectrumcomputing/lsf/conf
_ModuleTable004_=ZGV2L21vZHVsZWZpbGVzL3NpdGUvbGludXg
↪ tcmhlbDctcHBjNjRsZS9Db3JlL3hsLzE2LjEuMS1iZXRhNi5
↪ sdWEiLFsiZnVsbE5hbWUiXT0ieGwvMTYuMS4xLWJldGE2Iix
↪ bImxvYWRPcmRlciJdPTEscHJvcFQ9e30sWyJzdGF0dXMiXT0
↪ iYWN0aXZlIixbInVzZXJOYW1lIl09InhsIix9LH0sbXBhdGh
↪ BPXsiL2F1dG9mcy9uY2NzLXN2bTFfc3cvc3VtbWl0ZGV2L21
↪ vZHVsZWZpbGVzL3NpdGUvbGludXgtcmhlbDctcHBjNjRsZS9
↪ zcGVjdHJ1bS1tcGkvMTAuMi4wLjctMjAxODA4MzAtZXlvN3p
↪ 4bS94bC8xNi4xLjEtYmV0YTYiLCIvc3cvc3VtbWl0ZGV2L21
↪ vZHVsZWZpbGVzL3NpdGUvbGludXgtcmhlbDctcHBjNjRsZS9
↪ 4bC8xNi4xLjEtYmV0YTYiLCIvc3cvc3VtbWl0ZGV2L21v
CPATH=/autofs/nccs-svm1_sw/summitdev/.swci/1-compute
↪ /opt/spack/20180914/linux-rhel7-ppc64le/xl-16.1.
↪ 1-beta6/spectrum-mpi-10.2.0.7-20180830-eyo7zxm2p
↪ iusmyffr3iytmgwdacl67ju/include
HOSTTYPE=ppc64le-linux
OLCF_XLMASS_ROOT=/sw/summitdev/xl/16.1.1-beta6/xlmas
↪ s/9.1.1
OLCF_LMOD_ROOT=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4
↪ .8.5
PROJWORK=/gpfs/alpine/proj-shared
LMOD_FAMILY_MPI_VERSION=10.2.0.7-20180830
LMOD_PREPEND_BLOCK=normal
NLSPATH=/sw/summitdev/xl/16.1.1-beta6/msg/en_US/%N:/
↪ sw/summitdev/xl/16.1.1-beta6/xlC/16.1.1/msg/en_U
↪ S/%N:/sw/summitdev/xl/16.1.1-beta6/xlf/16.1.1/ms
↪ g/en_US/%N
_ModuleTable001_=X01vZHVsZVRhYmxlXz17WyJNVHZlcnNpb24
↪ iXT0zLFsiY19yZWJ1aWxkVGltZSJdPWZhbHNlLFsiY19zaG9
↪ ydFRpbWUiXT1mYWxzZSxkZXB0aFQ9e30sZmFtaWx5PXtbImN
↪ vbXBpbGVyIl09InhsIixbIm1waSJdPSJzcGVjdHJ1bS1tcGk
↪ iLH0sbVQ9e0RlZkFwcHM9e1siZm4iXT0iL3N3L3N1bW1pdGR
↪ ldi9tb2R1bGVmaWxlcy9zaXRlL2xpbnV4LXJoZWw3LXBwYzY
↪ 0bGUvQ29yZS9EZWZBcHBzLmx1YSIsWyJmdWxsTmFtZSJdPSJ
↪ EZWZBcHBzIixbImxvYWRPcmRlciJdPTYscHJvcFQ9e30sWyJ
↪ zdGF0dXMiXT0iYWN0aXZlIixbInVzZXJOYW1lIl09IkRlZkF
↪ wcHMiLH0saHNpPXtbImZuIl09Ii9zdy9zdW1taXRkZXYvbW9
↪ kdWxlZmlsZXMvc2l0ZS9saW51eC1yaGVsNy1wcGM2NGxl
MAIL=/var/spool/mail/USER
PATH=/sw/sources/lsf-tools/2.0/summitdev/bin:/sw/sum
↪ mitdev/xalt/1.1.3/bin:/sw/sources/hpss/bin:/auto
↪ fs/nccs-svm1_sw/summitdev/.swci/1-compute/opt/sp
↪ ack/20180914/linux-rhel7-ppc64le/xl-16.1.1-beta6
↪ /spectrum-mpi-10.2.0.7-20180830-eyo7zxm2piusmyff
↪ r3iytmgwdacl67ju/bin:/sw/summitdev/xl/16.1.1-bet
↪ a6/xlC/16.1.1/bin:/sw/summitdev/xl/16.1.1-beta6/
↪ xlf/16.1.1/bin:/opt/ibm/spectrumcomputing/lsf/10
↪ .1/linux3.10-glibc2.17-ppc64le-csm/etc:/opt/ibm/
↪ spectrumcomputing/lsf/10.1/linux3.10-glibc2.17-p
↪ pc64le-csm/bin:/usr/local/bin:/usr/bin:/usr/loca
↪ l/sbin:/usr/sbin:/opt/ibm/flightlog/bin:/opt/ibu
↪ tils/bin:/opt/ibm/spectrum_mpi/jsm_pmix/bin:/opt
↪ /puppetlabs/bin:/usr/lpp/mmfs/bin:/ccs/home/USER
↪ /.local/bin:/ccs/home/USER/bin
```

```
OPAL_LIBDIR=/autofs/nccs-svm1_sw/summitdev/.swci/1-c
↪ ompute/opt/spack/20180914/linux-rhel7-ppc64le/xl
↪ -16.1.1-beta6/spectrum-mpi-10.2.0.7-20180830-eyo7
↪ zxm2piusmyffr3iytmgwdacl67ju/lib
OLCF_XLC_ROOT=/sw/summitdev/xl/16.1.1-beta6/xlC/16.1
↪ .1
LMOD_SETTARG_CMD=:
OMPI_DIR=/autofs/nccs-svm1_sw/summitdev/.swci/1-comp
↪ ute/opt/spack/20180914/linux-rhel7-ppc64le/xl-16
↪ .1.1-beta6/spectrum-mpi-10.2.0.7-20180830-eyo7zx
↪ m2piusmyffr3iytmgwdacl67ju
PWD=/ccs/home/USER
_LMFILES_=/sw/summitdev/modulefiles/site/linux-rhel7
↪ -ppc64le/Core/xl/16.1.1-beta6.lua:/sw/summitdev/m
↪ odulefiles/site/linux-rhel7-ppc64le/xl/16.1.1-be
↪ ta6/spectrum-mpi/10.2.0.7-20180830.lua:/sw/summi
↪ tdev/modulefiles/site/linux-rhel7-ppc64le/Core/h
↪ si/5.0.2.p5.lua:/sw/summitdev/modulefiles/site/l
↪ inux-rhel7-ppc64le/Core/xalt/1.1.3.lua:/sw/summi
↪ tdev/modulefiles/site/linux-rhel7-ppc64le/Core/l
↪ sf-tools/2.0.lua:/sw/summitdev/modulefiles/site/
↪ linux-rhel7-ppc64le/Core/DefApps.lua
OLCF_MODULEPATH_ROOT=/sw/summitdev/modulefiles
MODULEPATH=/autofs/nccs-svm1_sw/summitdev/modulefile
↪ s/site/linux-rhel7-ppc64le/spectrum-mpi/10.2.0.7
↪ -20180830-eyo7zxm/xl/16.1.1-beta6:/sw/summitdev/m
↪ odulefiles/site/linux-rhel7-ppc64le/xl/16.1.1-be
↪ ta6:/sw/summitdev/modulefiles/site/linux-rhel7-p
↪ pc64le/Core:/sw/summitdev/modulefiles/core:/sw/s
↪ ummitdev/lmod/7.4.0/rhel7.2_gnu4.8.5/modulefiles
↪ /Linux:/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.5
↪ /modulefiles/Core:/sw/summitdev/lmod/7.4.0/rhel7
↪ .2_gnu4.8.5/lmod/lmod/modulefiles/Core
LOADEDMODULES=xl/16.1.1-beta6:spectrum-mpi/10.2.0.7-
↪ 20180830:hsi/5.0.2.p5:xalt/1.1.3:lsf-tools/2.0:D
↪ efApps
_ModuleTable_Sz_=6
LMOD_SYSTEM_NAME=summitdev
OLCF_XLF_ROOT=/sw/summitdev/xl/16.1.1-beta6/xlf/16.1
↪ .1
_ModuleTable005_=ZHVsZWZpbGVzL3NpdGUvbGludXgtcmhlbDc
↪ tcHBjNjRsZS9Db3JlIiwiL3N3L3N1bW1pdGRldi9tb2R1bGV
↪ maWxlcy9jb3JlIiwiL3N3L3N1bW1pdGRldi9sbW9kLzcuNC4
↪ wL3JoZWw3LjJfZ251NC44LjUvbW9kdWxlZmlsZXMvTGludXg
↪ iLCIvc3cvc3VtbWl0ZGV2L2xtb2QvNy40LjAvcmhlbDcuMl9
↪ nbnU0LjguNS9tb2R1bGVmaWxlcy9Db3JlIiwiL3N3L3N1bW1
↪ pdGRldi9sbW9kLzcuNC4wL3JoZWw3LjJfZ251NC44LjUvbG1
↪ vZC9sbW9kL21vZHVsZWZpbGVzL0NvcmUiLH0sWyJzeXN0ZW1
↪ CYXNlTVBBVEgiXT0iL3N3L3N1bW1pdGRldi9sbW9kLzcuNC4
↪ wL3JoZWw3LjJfZ251NC44LjUvbW9kdWxlZmlsZXMvTGludXg
↪ 6L3N3L3N1bW1pdGRldi9sbW9kLzcuNC4wL3JoZWw3LjJf
LMOD_CMD=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.5/l
↪ mod/lmod/libexec/lmod
LSF_BINDIR=/opt/ibm/spectrumcomputing/lsf/10.1/linux
↪ 3.10-glibc2.17-ppc64le-csm/bin
WORLDWORK=/gpfs/alpine/world-shared
MEMBERWORK=/gpfs/alpine/scratch/USER
SHLVL=2
HOME=/ccs/home/USER
OMPI_CC=/sw/summitdev/xl/16.1.1-beta6/xlC/16.1.1/bin
↪ /xlc_r
OSTYPE=linux
_ModuleTable002_=L0NvcmUvaHNpLzUuMC4yLnA1Lmx1YSIsWyJ
↪ mdWxsTmFtZSJdPSJoc2kvNS4wLjIucDUiLFsibG9hZE9yZGV
↪ yIl09Myxwcm9wVD17fSxbInN0YXR1cyJdPSJhY3RpdmUiLFs
↪ idXNlck5hbWUiXT0iaHNpIix9LFsibHNmLXRvb2xzIl09e1s
↪ iZm4iXT0iL3N3L3N1bW1pdGRldi9tb2R1bGVmaWxlcy9zaXR
↪ lL2xpbnV4LXJoZWw3LXBwYzY0bGUvQ29yZS9sc2YtdG9vbHM
↪ vMi4wLmx1YSIsWyJmdWxsTmFtZSJdPSJsc2YtdG9vbHMvMi4
↪ wIixbImxvYWRPcmRlciJdPTUscHJvcFQ9e30sWyJzdGF0dXM
↪ iXT0iYWN0aXZlIixbInVzZXJOYW1lIl09ImxzZi10b29scyI
↪ sfSxbInNwZWN0cnVtLW1waSJdPXtbImZuIl09Ii9zdy9zdW1
↪ taXRkZXYvbW9kdWxlZmlsZXMvc2l0ZS9saW51eC1yaGVs
OLCF_HSI_ROOT=/sw/sources/hpss
BASH_ENV=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.5/l
↪ mod/lmod/init/bash
VENDOR=apple
LESS=-deij4R
PYTHONPATH=/sw/summitdev/xalt/1.1.3/site:/sw/summitd
↪ ev/xalt/1.1.3/libexec
MPI_ROOT=/autofs/nccs-svm1_sw/summitdev/.swci/1-comp
↪ ute/opt/spack/20180914/linux-rhel7-ppc64le/xl-16
↪ .1.1-beta6/spectrum-mpi-10.2.0.7-20180830-eyo7zx
↪ m2piusmyffr3iytmgwdacl67ju
MACHTYPE=ppc64le
LOGNAME=USER
OLCF_SPECTRUM_MPI_ROOT=/autofs/nccs-svm1_sw/summitde
↪ v/.swci/1-compute/opt/spack/20180914/linux-rhel7
↪ -ppc64le/xl-16.1.1-beta6/spectrum-mpi-10.2.0.7-20
↪ 180830-eyo7zxm2piusmyffr3iytmgwdacl67ju
CVS_RSH=ssh
XLSF_UIDDIR=/opt/ibm/spectrumcomputing/lsf/10.1/linu
↪ x3.10-glibc2.17-ppc64le-csm/lib/uid
SSH_CONNECTION=160.91.202.153 47540 128.219.141.227
↪ 22
MODULESHOME=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.
↪ 5/lmod/lmod
OMP_NUM_THREADS=16
OMPI_CXX=/sw/summitdev/xl/16.1.1-beta6/xlC/16.1.1/bi
↪ n/xlc++_r
LESSOPEN=||/usr/bin/lesspipe.sh %s
__Init_Default_Modules=1
LMOD_MPI_VERSION=10.2.0.7-20180830-eyo7zxm
LMOD_FAMILY_COMPILER=xl
LMOD_FULL_SETTARG_SUPPORT=no
XL_LINKER=/sw/summitdev/xalt/1.1.3/bin/ld
XALT_OLCF=1
CMAKE_PREFIX_PATH=/autofs/nccs-svm1_sw/summitdev/.sw
↪ ci/1-compute/opt/spack/20180914/linux-rhel7-ppc6
↪ 4le/xl-16.1.1-beta6/spectrum-mpi-10.2.0.7-201808
↪ 30-eyo7zxm2piusmyffr3iytmgwdacl67ju
XDG_RUNTIME_DIR=/run/user/11670
```

```
LMOD_Priority_PATH=/sw/sources/lsf-tools/2.0/summitd
↪  ev/bin:-9999;/sw/summitdev/xalt/1.1.3/bin:-9999
_ModuleTable006_=Z251NC44LjUvbW9kdWxlZmlsZXMvQ29yZTo
↪  vc3cvc3VtbWl0ZGV2L2xtb2QvNy40LjAvcmhlbDcuMl9nbnU
↪  0LjguNS9sbW9kL2xtb2QvbW9kdWxlZmlsZXMvQ29yZSIsfQ==
LMOD_DIR=/sw/summitdev/lmod/7.4.0/rhel7.2_gnu4.8.5/l
↪  mod/lmod/libexec
LSF_ENVDIR=/opt/ibm/spectrumcomputing/lsf/conf
LMOD_FAMILY_MPI=spectrum-mpi
LMOD_COLORIZE=yes
_=/usr/bin/env
LSB Version:        :core-4.1-noarch:core-4.1-ppc64le
Distributor ID:        RedHatEnterpriseServer
Description:        Red Hat Enterprise Linux Server
↪  release 7.5 (Maipo)
Release:        7.5
Codename:        Maipo
Linux summitdev-login1 3.10.0-862.14.4.el7.ppc64le #1
↪  SMP Fri Sep 21 09:14:00 UTC 2018 ppc64le ppc64le
↪  ppc64le GNU/Linux
Architecture:          ppc64le
Byte Order:            Little Endian
CPU(s):                160
On-line CPU(s) list:   0-159
Thread(s) per core:    8
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Model:                 1.0 (pvr 004c 0100)
Model name:            POWER8NVL (raw), altivec
↪  supported
CPU max MHz:           4023.0000
CPU min MHz:           2061.0000
L1d cache:             64K
L1i cache:             32K
L2 cache:              512K
L3 cache:              8192K
NUMA node0 CPU(s):     0-79
NUMA node1 CPU(s):     80-159
MemTotal:       266798272 kB
MemFree:        143127168 kB
MemAvailable:   193878272 kB
Buffers:               0 kB
Cached:         90067136 kB
SwapCached:            0 kB
Active:         49730432 kB
Inactive:       56105792 kB
Active(anon):   19571904 kB
Inactive(anon): 36473536 kB
Active(file):   30158528 kB
Inactive(file): 19632256 kB
Unevictable:     4110464 kB
Mlocked:         4110464 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             128 kB
AnonPages:      19889472 kB
Mapped:          4591616 kB
Shmem:          40276352 kB
Slab:            5640192 kB
SReclaimable:    2401728 kB
SUnreclaim:      3238464 kB
KernelStack:       47984 kB
PageTables:        29440 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    133399104 kB
Committed_AS:   62669888 kB
VmallocTotal:   8589934592 kB
VmallocUsed:     5147200 kB
VmallocChunk:   8584406720 kB
HardwareCorrupted:     0 kB
AnonHugePages:    180224 kB
CmaTotal:       13434880 kB
CmaFree:        13434880 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:      16384 kB
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda       8:0    1        0 disk
sdb       8:16   1        0 disk
sdc       8:32   1        0 disk
sdd       8:48   1        0 disk
sde       8:64   1        0 disk
sdf       8:80   1        0 disk
sdg       8:96   1        0 disk
sdh       8:112  1        0 disk
sr0      11:0    1 1024M  0 rom
sr1      11:1    1 1024M  0 rom
sr2      11:2    1 1024M  0 rom
sr3      11:3    1 1024M  0 rom
nvme0n1 259:0    0  1.5T  0 disk
Thu Apr 11 03:16:14 2019
+-----------------------------------------------
↪  -------------------------+
| NVIDIA-SMI 396.44                 Driver Version:
↪  396.44                    |
|-------------------------------+----------------------
↪  ---+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A
↪  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage
↪  | GPU-Util  Compute M. |
|===============================+==================
↪  ===+======================|
|   0  Tesla P100-SXM2...  On   | 00000002:01:00.0 Off
↪  |                    0 |
```

```
| N/A   32C    P0    30W / 300W |      0MiB / 16280MiB
↪  |      0%   E. Process |
+-------------------------------+----------------------┐
↪  ---+-----------------------+
|   1  Tesla P100-SXM2...  On   | 00000003:01:00.0 Off
↪  |                    0 |
| N/A   28C    P0    31W / 300W |      0MiB / 16280MiB
↪  |      0%   E. Process |
+-------------------------------+----------------------┐
↪  ---+-----------------------+
|   2  Tesla P100-SXM2...  On   | 00000006:01:00.0 Off
↪  |                    0 |
| N/A   33C    P0    29W / 300W |      0MiB / 16280MiB
↪  |      0%   E. Process |
+-------------------------------+----------------------┐
↪  ---+-----------------------+
|   3  Tesla P100-SXM2...  On   | 00000007:01:00.0 Off
↪  |                    0 |
| N/A   30C    P0    30W / 300W |      0MiB / 16280MiB
↪  |      0%   E. Process |
+-------------------------------+----------------------┐
↪  ---+-----------------------+

+-----------------------------------------------------┐
↪  ---------------------------+
| Processes:
↪             GPU Memory |
|  GPU       PID   Type   Process name
↪              Usage      |
|=====================================================┐
↪  ===========================|
|  No running processes found
↪                            |
+-----------------------------------------------------┐
↪  ---------------------------+
0000:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0000:01:00.0 Ethernet controller: Broadcom Limited
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet (rev
↪  10)
0000:01:00.1 Ethernet controller: Broadcom Limited
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet (rev
↪  10)
0000:01:00.2 Ethernet controller: Broadcom Limited
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet (rev
↪  10)
0000:01:00.3 Ethernet controller: Broadcom Limited
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet (rev
↪  10)
0001:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0001:01:00.0 Non-Volatile memory controller: HGST,
↪  Inc. Ultrastar SN100 Series NVMe SSD (rev 05)
0002:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0002:01:00.0 3D controller: NVIDIA Corporation
↪  GP100GL [Tesla P100 SXM2 16GB] (rev a1)
0003:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0003:01:00.0 3D controller: NVIDIA Corporation
↪  GP100GL [Tesla P100 SXM2 16GB] (rev a1)
0004:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0004:01:00.0 Infiniband controller: Mellanox
↪  Technologies MT27700 Family [ConnectX-4]
0004:01:00.1 Infiniband controller: Mellanox
↪  Technologies MT27700 Family [ConnectX-4]
0005:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0005:01:00.0 PCI bridge: PLX Technology, Inc. PEX
↪  8718 16-Lane, 5-Port PCI Express Gen 3 (8.0 GT/s)
↪  Switch (rev ab)
0005:02:01.0 PCI bridge: PLX Technology, Inc. PEX
↪  8718 16-Lane, 5-Port PCI Express Gen 3 (8.0 GT/s)
↪  Switch (rev ab)
0005:02:02.0 PCI bridge: PLX Technology, Inc. PEX
↪  8718 16-Lane, 5-Port PCI Express Gen 3 (8.0 GT/s)
↪  Switch (rev ab)
0005:02:03.0 PCI bridge: PLX Technology, Inc. PEX
↪  8718 16-Lane, 5-Port PCI Express Gen 3 (8.0 GT/s)
↪  Switch (rev ab)
0005:02:04.0 PCI bridge: PLX Technology, Inc. PEX
↪  8718 16-Lane, 5-Port PCI Express Gen 3 (8.0 GT/s)
↪  Switch (rev ab)
0005:03:00.0 USB controller: Texas Instruments
↪  TUSB73x0 SuperSpeed USB 3.0 xHCI Host Controller
↪  (rev 02)
0005:04:00.0 SATA controller: Marvell Technology
↪  Group Ltd. 88SE9235 PCIe 2.0 x2 4-port SATA 6 Gb/s
↪  Controller (rev 11)
0005:05:00.0 PCI bridge: ASPEED Technology, Inc.
↪  AST1150 PCI-to-PCI Bridge (rev 03)
0005:06:00.0 VGA compatible controller: ASPEED
↪  Technology, Inc. ASPEED Graphics Family (rev 30)
0005:07:00.0 Ethernet controller: Broadcom Limited
↪  NetXtreme BCM5719 Gigabit Ethernet PCIe (rev 01)
0005:07:00.1 Ethernet controller: Broadcom Limited
↪  NetXtreme BCM5719 Gigabit Ethernet PCIe (rev 01)
0006:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0006:01:00.0 3D controller: NVIDIA Corporation
↪  GP100GL [Tesla P100 SXM2 16GB] (rev a1)
0007:00:00.0 PCI bridge: IBM POWER8 Host Bridge (PHB3)
0007:01:00.0 3D controller: NVIDIA Corporation
↪  GP100GL [Tesla P100 SXM2 16GB] (rev a1)
0008:00:00.0 Bridge: IBM Device 04ea
0008:00:00.1 Bridge: IBM Device 04ea
0008:00:01.0 Bridge: IBM Device 04ea
0008:00:01.1 Bridge: IBM Device 04ea
0009:00:00.0 Bridge: IBM Device 04ea
0009:00:00.1 Bridge: IBM Device 04ea
0009:00:01.0 Bridge: IBM Device 04ea
0009:00:01.1 Bridge: IBM Device 04ea
```