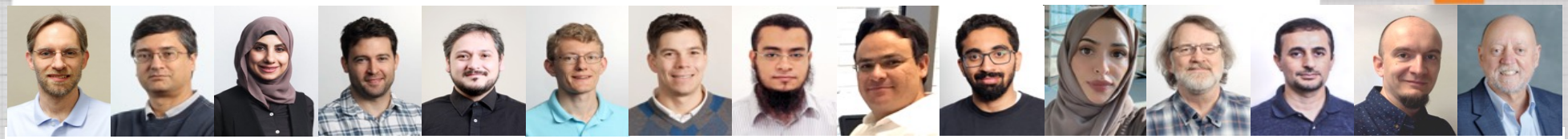
Tasking in the SLATE Dense Linear Algebra Library

Asim YarKhan (ICL/UTK)

16th Scheduling for Large Scale Systems Workshop

University of Tennessee Knoxville

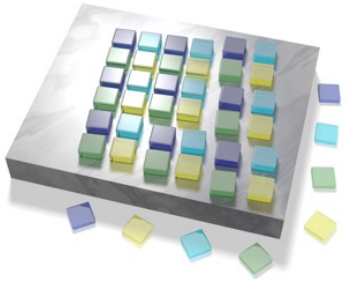
May 22-24, 2023



Mark Gates, Asim YarKhan, Dalal Sukkari, Sébastien Cayrols, Daniel Bielich, Neil Lindquist, Hartwig Anzt, Ahmad Abdelfattah, Kadir Akbudak, Mohammed Al Farhan, Rabab Al-omair, Anthony Castaldo, Ali Charara, Jakub Kurzak, Jack Dongarra

SLATE: Software for Linear Algebra Targeting Exascale

- Distributed, GPU-accelerated, dense linear algebra library
  - Target: Large-scale, multi-{GPU, core, memory}
  - Modern replacement for ScaLAPACK
- Functionality
  - BLAS (matrix multiplies, triangular solves), norm, cond, ...
  - Linear solves (Cholesky, Cholesky, symmetric indefinite)
  - Least squares (QR, Cholesky QR, LQ)
  - Hermitian eigenvalue, generalized Hermitian eigenvalue
  - Singular value decomposition (SVD)



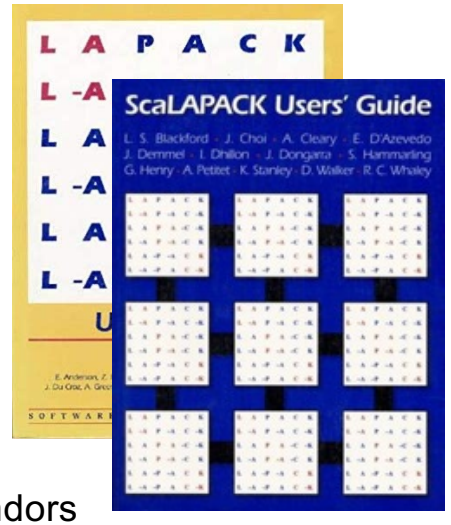
ORNL
IBM/NVIDIA

ORNL
HPE/AMD

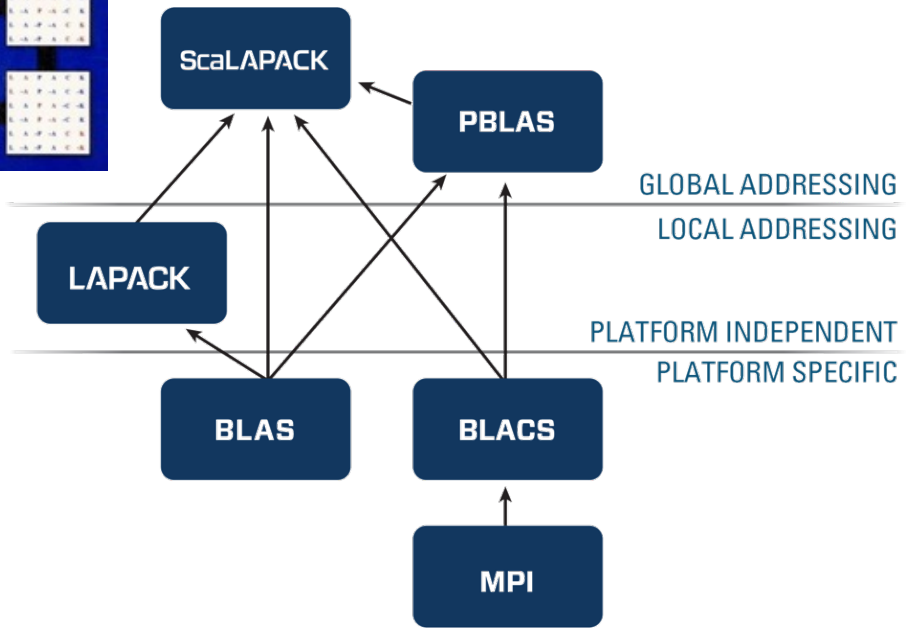
LBNL
HPE/AMD/NVIDIA

ANL
Intel/HPE

ScaLAPACK

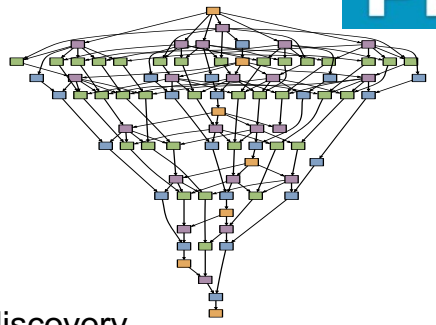


- Dense linear algebra library
  - **First released in 1995**
  - Robust, highly tested package
  - Asymptotically scalable
  - CPU performance is still hard to beat
  - De-facto standard, used by many vendors
- **No clear path to GPU support**
- Lessons for success
  - Longevity due to a dependence on core features
  - BLAS, LAPACK, PBLAS, BLACS / MPI
  - Components that provide abstract matrix operations
  - Vendors optimize these core components



SLATE

- Linear algebra based on tiled data - PLASMA library
  - Shared memory, full data dependency (Aik:R, Bkj:R, Cij:RW)
  - Initially used a static state transition table
  - Moved to an internal DAG runtime QUARK
  - Moved to OpenMP dependencies
  - Lessons …
    - Static state tables are hard. Full DAGs are costly and can block task discovery. Standards can perform well.
- GPU work – MAGMA,
- Distributed memory LA experience
  - ScaLAPACK, HPL, Parsec/DPLASMA, various PhD projects with DAGs.



PLASMA

QUARK

HPL

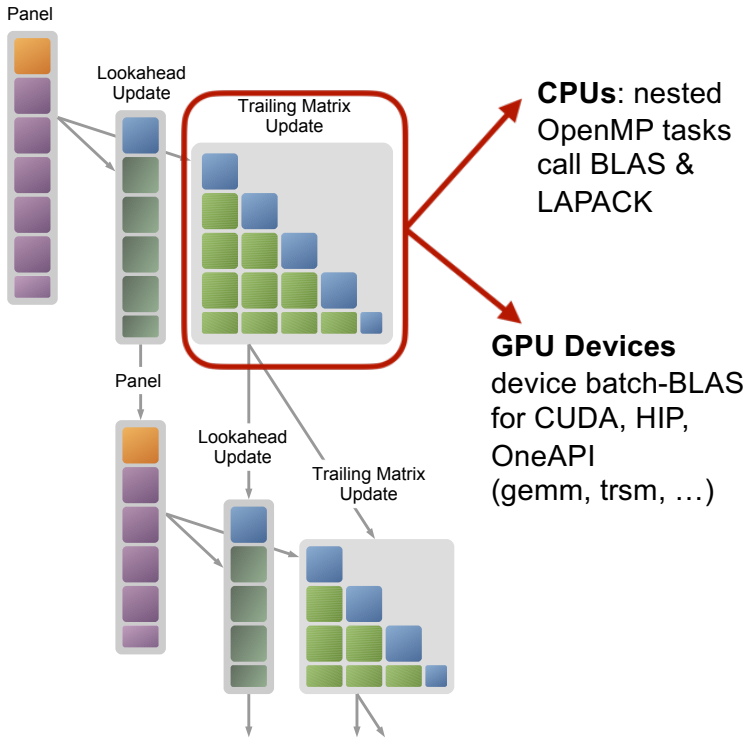
DPLASMA

PaRSEC

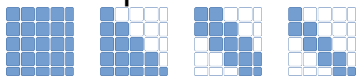
MAGMA

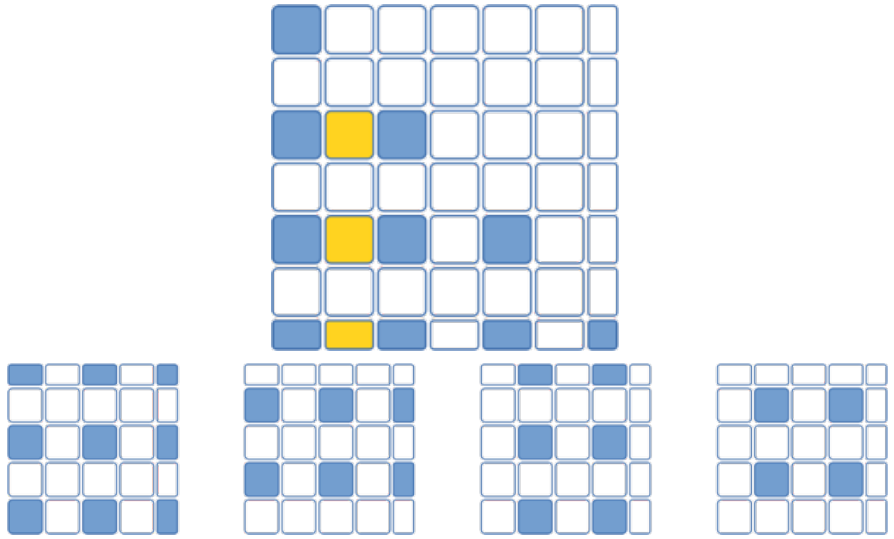
Design of SLATE

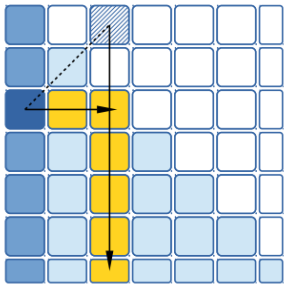
- **Data tiling**
  - Matrix – a “loose” collection of tiles
  - Any partitioning to nodes and devices
- **Dynamic tasking on nodes**
  - OpenMP for managing tasks and dependencies
- **Batch-execution for performance**
  - GPU: batch-BLAS / batch-GEMM
  - CPU: batch-BLAS or OpenMP tasking
  - Provided by BLASPP++ CPU/GPU wrappers
- **Lookahead**
  - Enabled by OpenMP data-dependence
  - Enable future panel without waiting for update operation



SLATE Matrix

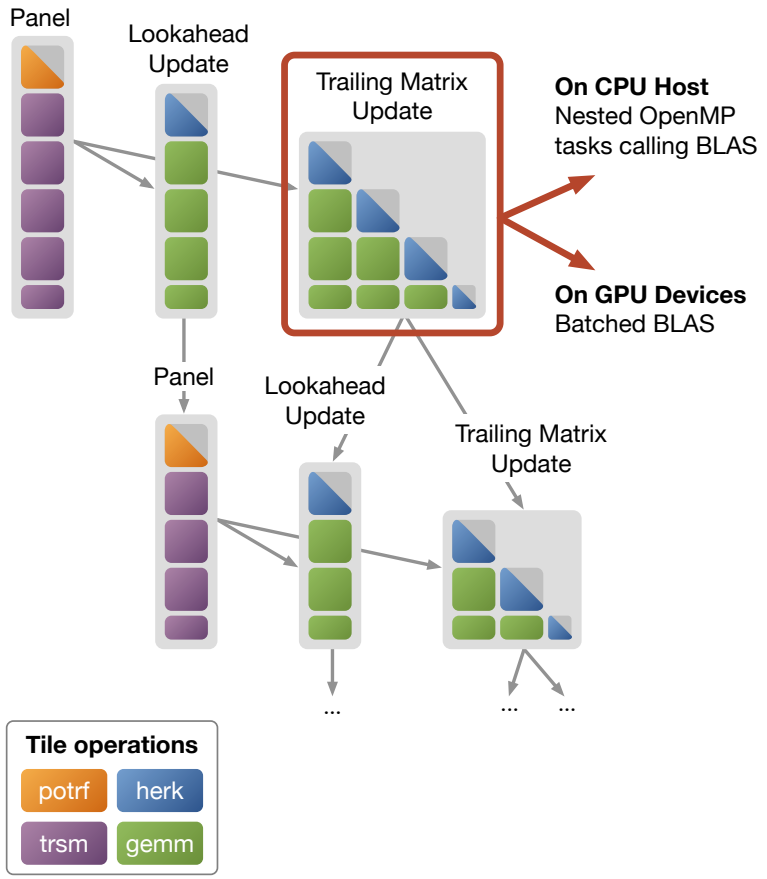
- Map from matrix/tile indices { i, j, device } to Tile
  - Map exists on all nodes
  - Local storage for tiles at each location
  - 2D block cyclic by default
    - Any mapping possible
  - Remote data is attached to map placeholder
  - Tiles can be strided (leading dimension)
    - Enables ScaLAPACK 2D block cyclic compatibility
  - Accommodates band and block sparse
    - No wasted memory space 
- Distributed algorithms
  - Remote tiles are communicated explicitly
  - Have a natural place to store remote tiles
  - Refer to remote tiles via known map indices





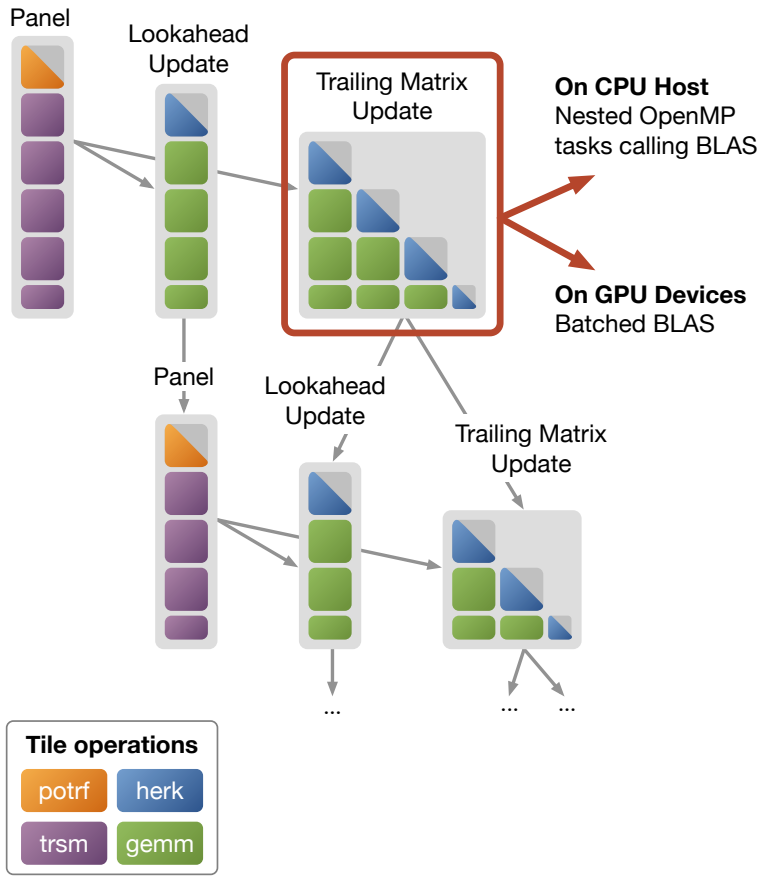
Multi-Level Tasking

- SLATE uses OpenMP as the local runtime scheduler
- Schedule aggregated-task using OpenMP dependencies
  - Results in O(n/nb) dependency task graph
- Within the aggregated-tasks, schedule work on tiles *without* dependencies
  - Or Batch BLAS on GPUs or CPUs
  - Either OpenMP nested parallelism on CPUS
- SLATE manages data movement (not the runtime)
  - SLATE moves data between nodes using MPI multicast of sets-of-tiles



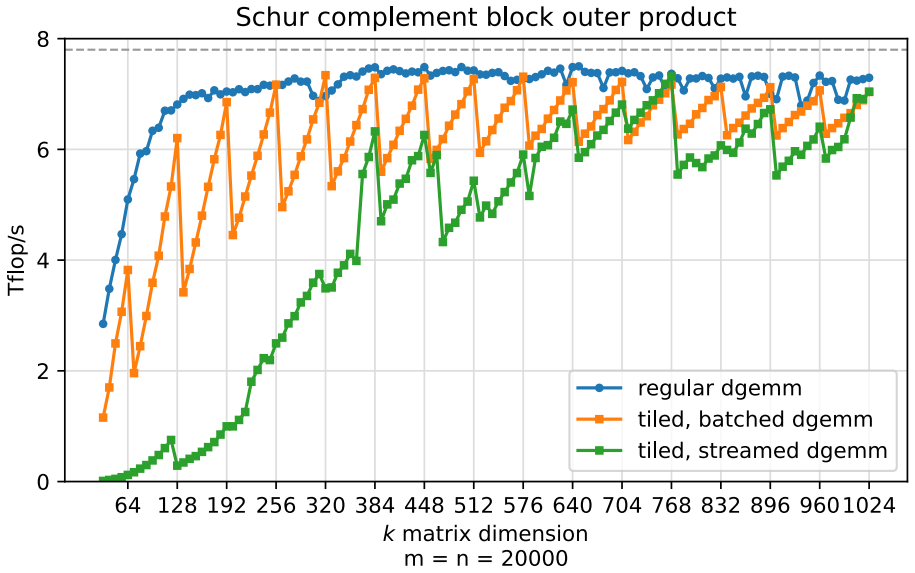
High-Level Algorithms

- **High-level algorithms are independent of architecture and data type**
  - For example, Cholesky/LU have one implementation, use C++ templates for data type & target
- **Low-level internal routines dispatch to target implementations**
  - internal::gemm has multiple target implementations:
    - **HostTask**: nested OpenMP tasks
    - **Device**: batched BLAS++ call on GPU



Batched BLAS in BLAS++

- SLATE algorithms expose parallelism by working on tiles of data
- But for GPUs, smaller tiles can have low performance
- Batched BLAS operations, implemented by GPU vendors, can extract high performance by batching/grouping operations on tiles of smaller sizes
  - Batch routines take a set of matrices and perform the same operation on all in parallel
- In SLATE, the trailing matrix update
  - Matrix-multiply of the panel block-column and a block-row to update the trailing matrix

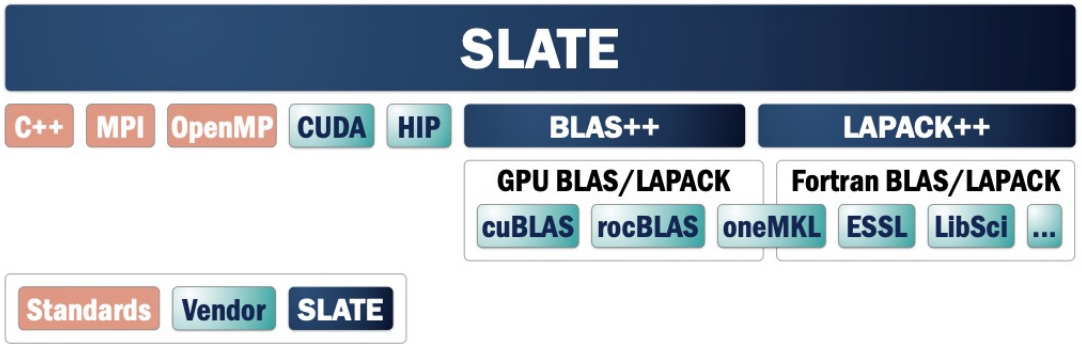


Volta V100 GPU, 7.8 Tflop/s peak

Block outer-product matrix multiply, implemented as single gemm, batched gemm on tiles, or individual tile gemms in separate streams, on NVIDIA V100 GPU.

Accelerator platforms

- Use BLAS++ as abstraction layer
  - cuBLAS backend (done)
  - hip/rocBLAS backend (done)
  - oneAPI backend (mostly complete)
- A few memory-bound GPU kernels are implemented within SLATE. (batched add tiles, scale tiles, norms of tiles)
  - Port to HIP using hipify (initial work done)
  - Intel OneAPI port to OpenMP device-offload in progress



BLAS++ and LAPACK++

- Simple, data-type templated interface to CPU routines
  - C++ wrappers around vendor optimized BLAS, LAPACK libraries
- BLAS++, LAPACK++ for GPU capabilities
  - Portable calls in SLATE using device queues
    - Supports batch-calls on device (batch-gemm, …)
    - cuBLAS backend and hip/rocBLAS backend
    - Intel OneAPI backend recently added backend
- LAPACK++ has some device routines
  - Solver queues and handles are similar to blas queues.
- Device memory management is done through BLAS++
- SLATE has very few memory-bound internal device kernels
  - Matrix norm, set, add, copy, scale (CUDA , hipified)
  - On SYCL we are using OpenMP device-offload for these kernels

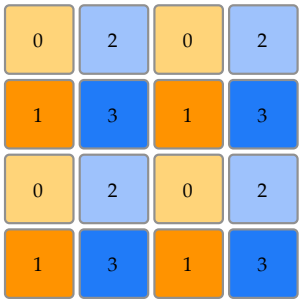
```
// BLAS++ call on CPU. x is in CPU memory.
blas::scal( n, alpha, x, incx );

// BLAS++ call on GPU. dx is in GPU device memory.
blas::scal( n, alpha, dx, incx, queue );
```

SLATE

Distributed, GPU-accelerated, dense linear algebra library.

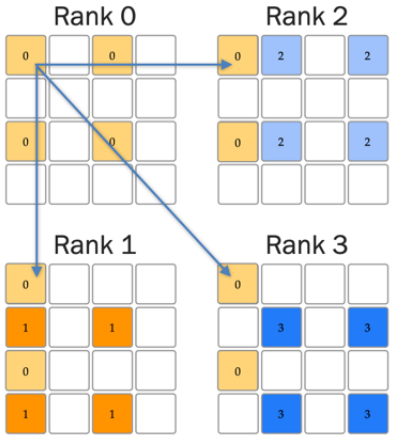
Modern replacement for ScaLAPACK.

https://icl.utk.edu/slate/



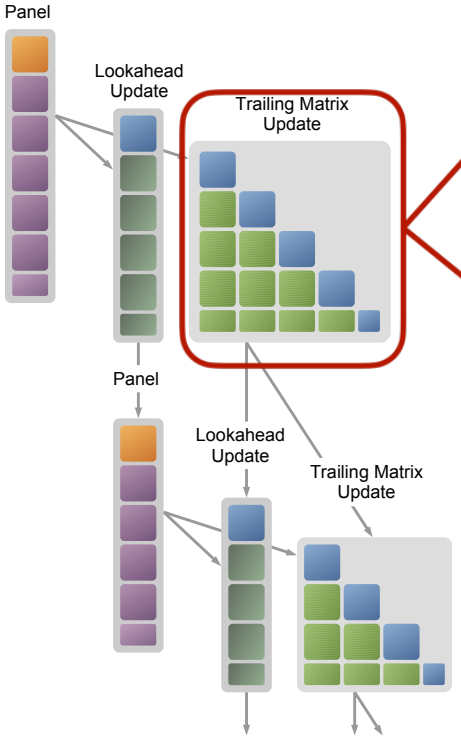
Matrices are composed of tiles of data distributed over processes

Each node has Map{i, j, device} to tile*



Algorithms are expressed as matrix operations on the tiles

Data is copied/referenced in local matrix structures



CPUs: nested OpenMP tasks call BLAS & LAPACK

GPU Devices device batch-BLAS for CUDA/HIP (gemm, trsm, …)

SLATE aggregates operations using panel + lookahead (for overlapping/hiding) + update operations.
OpenMP manages dependencies between aggregated task-groups.

SLATE Coverage

BLAS, etc. ($C = AB$, ...)

	ScaLAPACK	SLATE
Level 1 PBLAS	✔	✘
Level 2 PBLAS	✔	some
Level 3 PBLAS	✔	✔
Matrix norms	✔	✔
Test matrix generation	✔	✔

Linear systems ($Ax = b$)

	ScaLAPACK	SLATE
LU (partial pivoting)	✔	✔
LU, band (pp)	✔	✔
LU (non-pivoting)	✘	✔
Cholesky	✔	✔
Cholesky, band	✔	✔
Symmetric Indefinite (Aasen)	✘	✔ (CPU)
Mixed precision	✘	✔
Inverses (LU, Cholesky)	✔	✔
Condition estimate	✔	✔

Least squares ($Ax \cong b$)

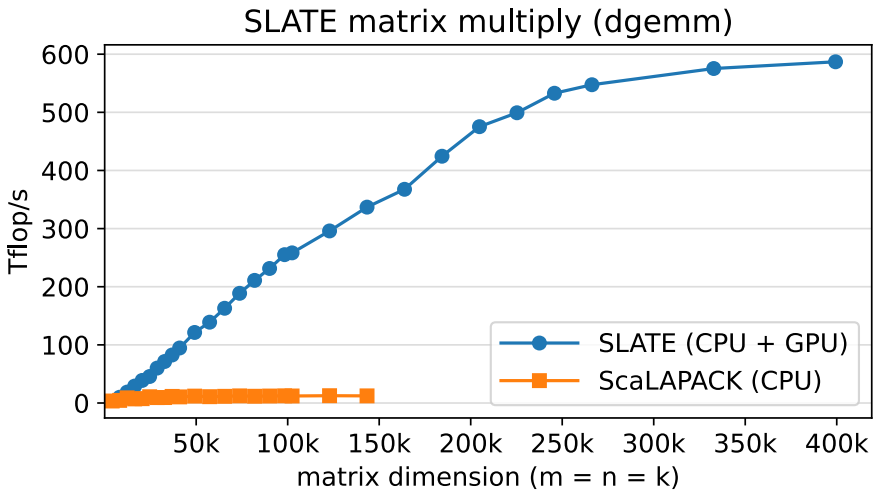
	ScaLAPACK	SLATE
QR	✔	✔
LQ	✔	✔
Least squares solver	✔	✔

SVD, eigenvalues ($A = U\Sigma V^{\mathrm{H}}$, $Ax = \lambda x$)

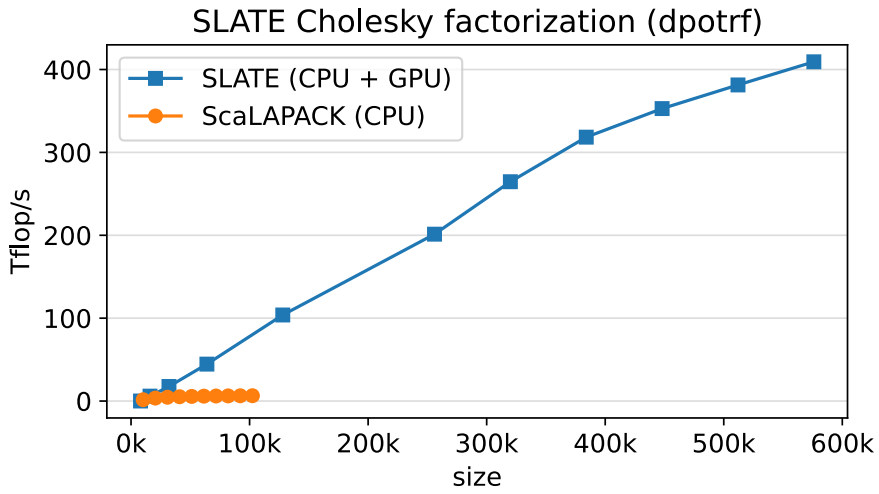
	ScaLAPACK	SLATE
SVD	✔	✔ (vectors 2023)
Symmetric eigenvalues	✔	✔
Generalized sym. eigenvalues	✔	✔
Divide and conquer	✔ (eig)	2023
Polar decomposition (QDWH)	✘	✔
Non-symmetric eigenvalues	parts	Post ECP

Matrix Multiply and Cholesky on Summit

47x GPU speedup
77% of peak



37x GPU speedup
53% of peak

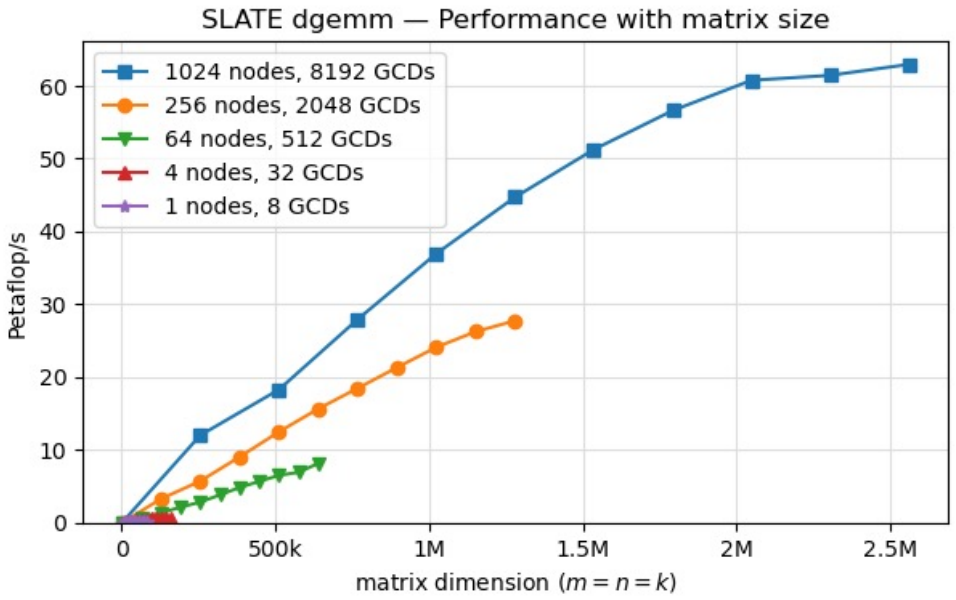


Summit: 16 nodes on Summit, 96 NVIDIA Volta V100 GPUs, 68
CPU+GPU peak 765 Tflop/s

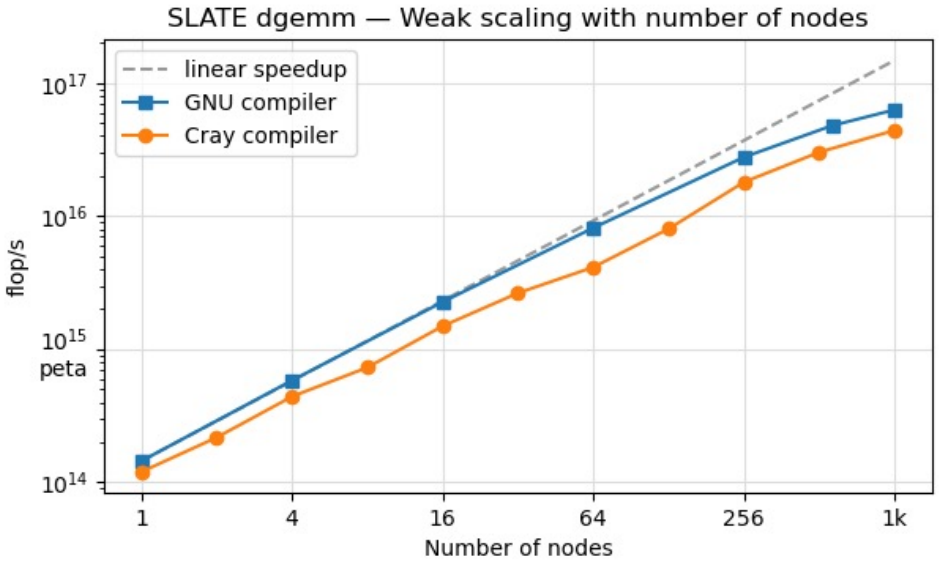
Summit: 16 nodes, 96 NVIDIA Volta V100, 672 cores IBM POWER9
CPU + GPU peak 765 Tflop/s

Single node GPU speedup similar: 50x for dgemm, 25x for dpotrf (limited problem size)

GEMM on Frontier

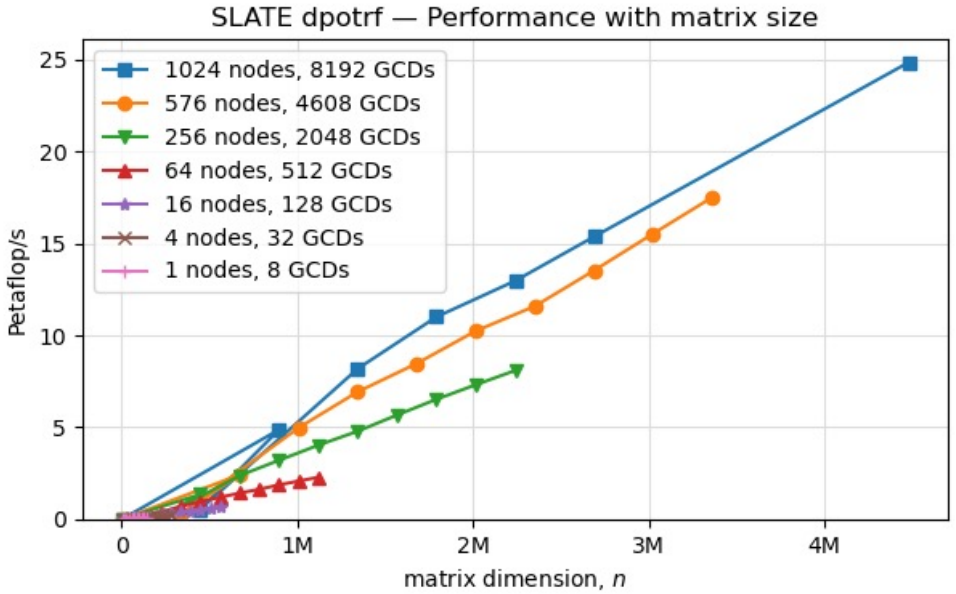


Frontier: 4 AMD MI250X GPUs (8 GCDs) per node; 1 MPI process per GCD; GNU compiler

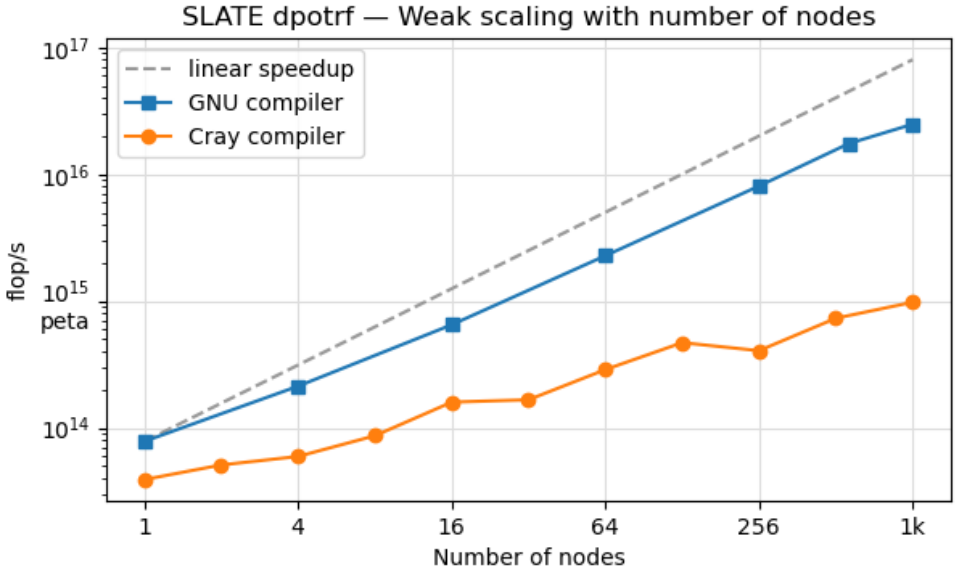


Frontier: 4 AMD MI250X GPUs (8 GCDs) per node; 1 MPI process per GCD
Weak scaling 80k x 80k per node

Cholesky on Frontier



Frontier: 4 AMD MI250X GPUs (8 GCDs) per node; 1 MPI process per GCD; GNU compiler



Frontier: 4 AMD MI250X GPUs (8 GCDs) per node; 1 MPI process per GCD
Weak scaling 140k x 140k per node

Productivity: C++ Templates and Abstractions

- Extensive use of C++ templates
- Algorithmic variants
- Matrix object abstractions
- Simplified API for calls (e.g. gemm)
- Reduced code size

Templates

- precisions
  - `float`
  - `double`
  - `std::complex<float>`
  - `std::complex<double>`
- targets
  - `Target::Host`
  - `Target::Devices`

Algorithmic variants via option parameter

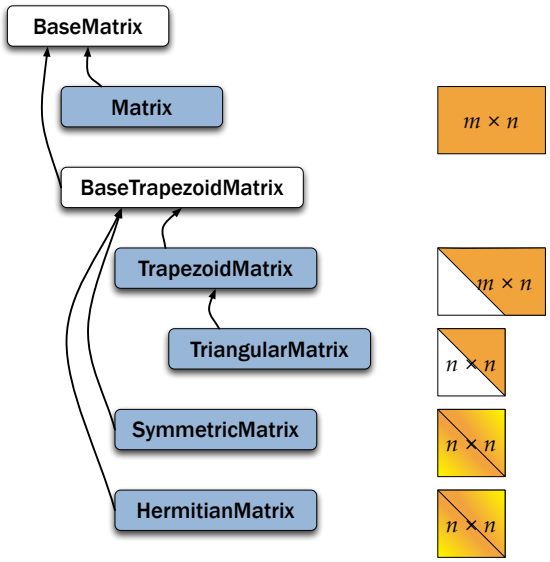
- Least squares dgels
  - Householder QR
  - Cholesky QR

Simplified API

- gemm (alpha, beta, A, B, C)

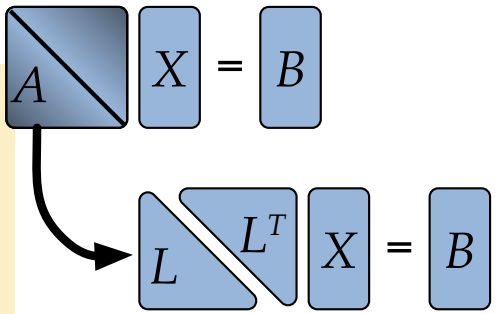
Standard Features

- inheritance
- overloading
- matrix hierarchy



Cholesky example using SLATE

```
// slate-tutorial linear_system_Cholesky.cc
template <typename scalar_type>
void test_cholesky()
{
    // MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, .. );
    int64_t n=1000, nrhs=100, nb=256, p=2, q=2;
    assert( mpi_size == p*q );
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, p, q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type>
        B( n, nrhs, nb, p, q, MPI_COMM_WORLD );
    A.insertLocalTiles();
    B.insertLocalTiles();
    random_matrix_diag_dominant( A );  // local support function
    random_matrix( B ); // local support function

    slate::potrf( A );
    slate::potrs( A, B );
}
```

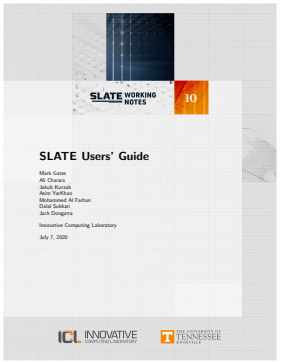


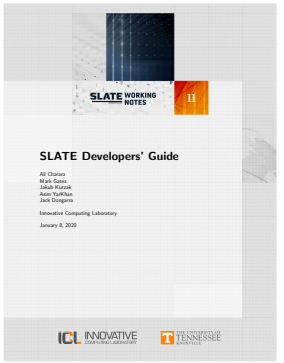
SLATE APIs

- Native C++ API
- C API
- ScaLAPACK wrapper API

Availability

- SLATE icl.utk.edu/slate
  - SLATE Github github.com/icl-utk-edu/slate
  - BLAS++ github.com/icl-utk-edu/blaspp
  - LAPACK++ github.com/icl-utk-edu/lapackpp
  - TestSweeper github.com/icl-utk-edu/testsweeper
  - Issue tracking & pull requests
- Modified BSD License



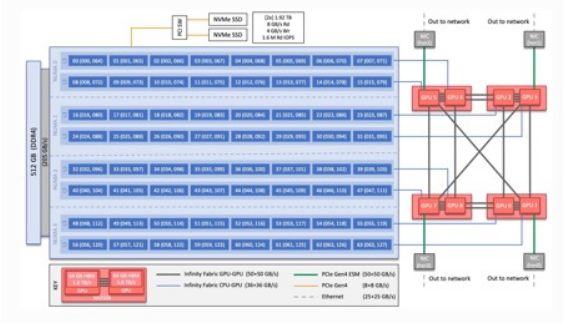


MAKE CMake git Spack xSDK E4S

ECP EXASCALE COMPUTING PROJECT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Challenge: HPL on Frontier



- HPL is a distributed LU benchmark with lookahead and GPU support
  - Thanks to AMD for providing source code to rocHPL
    - https://github.com/ROCmSoftwarePlatform/rocHPL
- HPL: 1.194 ExaFlops *1000*1000 / (9408 nodes * 4 GPU * 2 GCDs)
  - At scale 15 TF/GCD! (Theoretical peak 26.5 TF/GCD (double-precision))
- Change in Crusher/Frontier default thread binding reduced performance by 30%
  - Frontier reserved 1st core for system use
  - Needed to select all cores to fix binding (srun -S0)
- Carefully tuned code
  - Very specific binding of process-threads to hardware, block sizes, thread binding, oversubscription
  - Allows very specific overlap of GPU and CPU work
    - Panel done on the CPU using OpenMP threads
- Challenge: Make this hand-tuning happen automagically.


ICL
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY OF
TENNESSEE
KNOXVILLE