Performance Tuning SLATE

Mark Gates
Ali Charara
Asim YarKhan
Dalal Sukkari
Mohammed Al Farhan
Jack Dongarra

Innovative Computing Laboratory

January 2, 2020

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

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

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

Revision	Notes
01-2020	first publication

```
@techreport{gates2020performance,
  author={Gates, Mark and Charara, Ali and YarKhan, Asim
          and Sukkari, Dalal and Al Farhan, Mohammed and Dongarra, Jack},
  title={{SLATE} Working Note 14
         Performance Tuning SLATE},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2020},
  month={January},
  number={ICL-UT-XX-XX},
  note={revision 01-2020}
}
```

Contents

List of Figures

CHAPTER 1

Introduction

Software for Linear Algebra Targeting Exascale (SLATE) [1] is being developed as part of the Exascale Computing Project (ECP) [2], which is a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration (NNSA). SLATE will deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems, including GPU-accelerated systems as well as more traditional multi core–only systems.

SLATE provides coverage of existing LAPACK and ScaLAPACK functionality, including parallel implementations of Basic Linear Algebra Subroutines (BLAS), matrix norms, linear systems solvers, least squares solvers, and singular value and eigenvalue solvers. In this respect, SLATE will serve as a replacement for ScaLAPACK, which, after two decades of operation, cannot be adequately retrofitted for modern, GPU-accelerated architectures.

This working note focuses attention on several performance issues that existed in SLATE, and how these were resolved.

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

CHAPTER 2

Infrastructure Improvements

2.1 Asynchronous CPU ↔ Accelerator Copies

Layered infrastructure: SLATE's programming interfaces are composed of several layers. **Driver** routines solve an entire problem, such as a linear system $Ax = b$ (routines `gesv`, `posv`). Drivers in turn call **computational** routines to solve sub-problems, such as computing an LU factorization (`getrf`), or performing a matrix-matrix multiply (`gemm`). Computational routines in turn rely on a set 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 (without task dependencies). Internal routines provide device-specific implementations such as OpenMP nested tasks, parallel-for loops, or batch BLAS operations.

Memory spaces: SLATE operates on heterogeneous architectures, so the matrix data may originate or temporarily reside on any hardware memory space available in the system, such as host memory or GPU accelerator memory. 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.

Data coherency: For offload to GPU accelerators, SLATE implements a tile-based memory consistency model inspired by the MOSI cache coherency protocol. Data coherency is coordinated by the MOSI API, which provides routines to fetch one tile or a set of tiles into a memory space for:

- Reading (`tileGetForReading()`), or for
- Writing (`tileGetForWriting()`), or for
- Reading and holding (`tileGetAndHold()`).

SLATE allows a tile to co-exist in many memory spaces. A tile instance in a memory space may be in any of **M**odified, **I**nvalid, or **S**hared states. A tile may be orthogonally marked **O**nHold. However, a computational routine or internal routine need not be aware in which memory space the origin instance of a tile currently resides, nor in which memory space the most up-to-date instance currently resides. Instead, the MOSI API will implicitly fetch a tile's data from the most up-to-date tile instance upon calling any of the (`tileGet***()`) routines.

Bottlenecks found and solutions implemented

Synchronization after each tile copy → MOSI Async API: A `tileGet***()` operation fetches a data tile into the specified memory space if it doesn't already exist there or its data is outdated, using the CUDA stream of the destination device. Thus, it naturally synchronizes with that stream to ensure the tile's data would be available immediately afterwards.

However, we found that fetching a set of tiles does not require synchronizations after each tile copy. A single synchronization at the end of looping over the set of tiles suffices. An `Async MOSI API` has been introduced, thus, eliminating excessive synchronizations and allowing much faster data transfer rates. This improvement is reflected in the performance of almost all computational routines currently available in SLATE.

Pipelining data fetch of input matrices: An internal routine uses `tileGetForReading()` / `tileGetForWriting()` to fetch a set of tiles from each input or output matrix into local memory space. For example, `internal::gemm(alpha, A, B, beta, C)` sequentially calls:

```
1 A.tileGetForReading(A_tiles);
2 B.tileGetForReading(B_tiles);
3 C.tileGetForReading(C_tiles);
```

However, wrapping each `tileGet***()` call in an OpenMP task allows for parallel data fetching and a better saturation of the device communication stream. Such pipelining has been applied to all internal routines in SLATE.

Overlap of communication and computation with dedicated streams: For a proper overlap of communication and computation on devices, we fixed many problems in which CUDA computation kernels were carried on communication streams, as well as cases where data transfers from/to GPU memory were carried on CUDA computation streams. Dedicated

CUDA streams for computations and others for communications facilitated better overlap and pipelined work-flows and improved performance.

Updating tile origin: In computational routines, SLATE fetches tiles to the memory space where computations are performed. Often, tiles are fetched as workspace copies into memory spaces other than their origin. At the end of the computational routines, origin tile instances need to be updated to the latest copies. SLATE provide functions, within the MOSI API, to fetch tiles back to their origin. However, updating origin tiles was serialized among devices and synchronous for the same device. We have taskified updating origin tiles per device, and used the Async MOSI API to improve the throughput of data transfers.

Device workspace management: An origin instance of a tile is created at the first insertion of the tile into its matrix only on its MPI rank based on the matrix distribution. An origin tile instance may occupy a memory buffer provided by the user upon matrix creation, which is non-purgeable by SLATE, otherwise it occupies a memory buffer created by SLATE upon matrix creation, which is purged only at matrix destruction. When a tile is fetched into a different MPI rank, or fetched into a different memory space within the same MPI rank, it is stored in a workspace-memory. Workspace memory is allocated on a need basis. However, allocating device memory with `cudaMalloc` is expensive in that it is a synchronous operation. For that reason, SLATE provides a lightweight memory management construct that facilitates allocating a large enough memory pool that is used later to host individual tiles. Estimating the size of such a memory pool is a heuristic that can be optimized. It was based only on the maximum number of tiles that need to be updated on the devices. However, tiles that need to be hosted on the devices as read-only workspace were not accounted for. For a more consistent management of device workspace, the MOSI API has been updated to pre-allocate the workspace memory pool based on the number of tiles that are requested on each device. This update has removed extensive implicit synchronizations caused by allocating device memory at runtime, thus, improved the data transfer throughput.

Local data pre-fetch while MPI broadcasting: SLATE communicates tiles that are needed in other MPI ranks for the next iterations of each algorithm in the form of a broadcast. Such broadcasts are often carried during lookahead tasks, which allows overlapping local computations with communications needed for the next steps. A broadcast is built by scanning the sub-matrix that will be updated and broadcasting local tiles that are needed for this update. To improve the chances of overlapping communication and computation, we extended the MPI broadcasting mechanism to include broadcasting local tiles into device memory for the next steps. This extension pre-fetched data into devices' memories during computations, and drastically improved SLATE's overall data-transfer throughput.

2.2 Communication Improvements

In most of the algorithms in SLATE, communication is expressed as "broadcast" operations, where a list of tiles, labeled A, (often representing a row or column of the matrix) is broadcast to

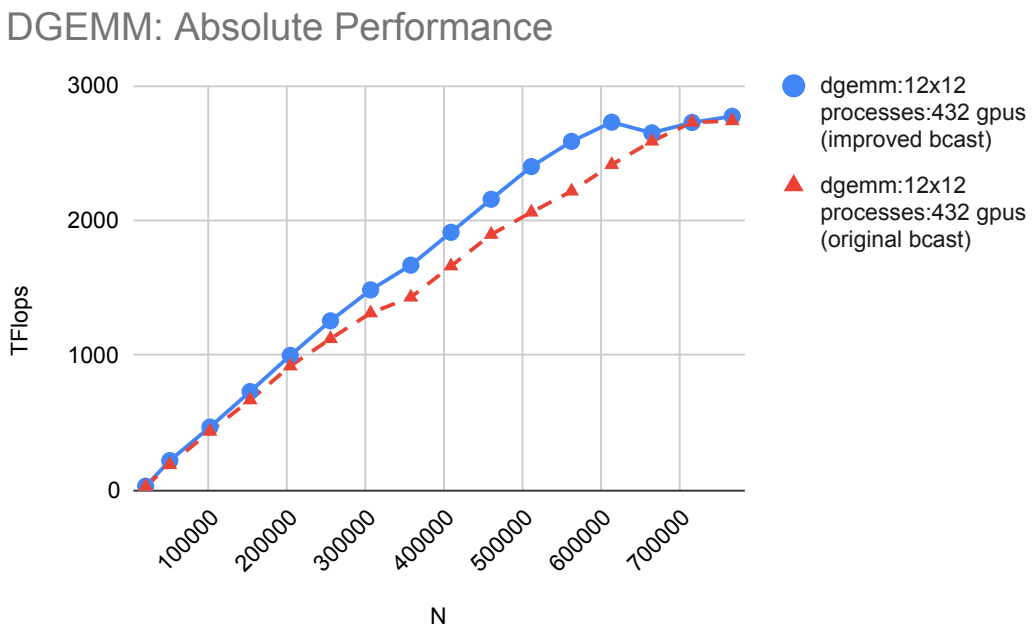

Figure 2.1: Performance of `dgemm` showing the effect of improving the SLATE broadcast operation. Using 72 nodes on Summit, one process per socket, 12×12 process grid, tile size 1024.

list of tiles, labeled C, that would need to use the A-tiles in the future. Some of the destination C-tiles may be local whereas others may be on remote distributed-memory nodes.

For each A-tile to be communicated, SLATE examines the list of receiver C-tiles to determine which non-local nodes will be participating in the broadcast. The data is broadcast using a multi-dimensional hypercube overlay network to the participating nodes. The nodes in the overlay network send or receive the A-tile and forward it as specified by the hypercube communication pattern.

The current improvement changes the hypercube communication forwarding action from a synchronous send to a sequence of non-blocking sends, followed by a wait-all. This localizes the code changes to a very controllable region, minimizing the impact at the algorithmic level.

Performance experiments were done using 72 nodes on Summit, ORNL's pre-exascale platform [1]. These 72 nodes were used by binding a process-per-socket (12×12 processes) with three NVIDIA V100 GPUs and one 21-core POWER9 processor per socket.

Using non-blocking sends resulted in significant performance improvement in the communication-bound regions of the performance curves, as shown in Figures 2.1 for the `dgemm` implementation and in Figure 2.2 for the `dpotrf` factorization.

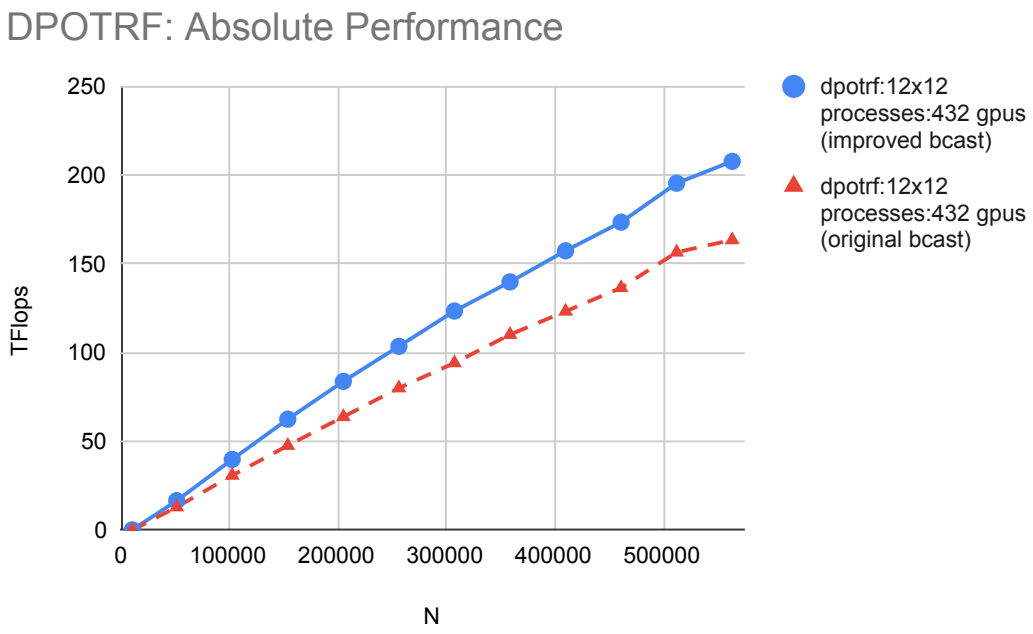

Figure 2.2: Performance of `potrf` showing the effect of improving the SLATE broadcast operation. Using 72 nodes on Summit, one process per socket, 12×12 process grid, tile size 320.

CHAPTER 3

QR, LQ, and GELS Improvements

SLATE uses a version of the communication avoiding QR (CAQR) factorization. In the QR panel, each node in the panel does a local QR factorization of its tiles, yielding a triangular tile on each node. Then the nodes collectively perform QR factorizations of pairs of the resulting triangles, in a binary tree fashion, using LAPACK's `tpqrt` routine, resulting in a single triangular tile for the entire panel. See [2] for more details.

When applying the trailing matrix update, the same structure must be followed: apply Householder reflectors from the local QR factorization using `unmqr`, then apply Householder reflectors from QR factorizations of pairs triangles using `ttmqr`, as illustrated in Figure 3.1. For each pair of block rows, the node owning the top row sends the row, tile-by-tile, to the node owning the bottom row; the two rows are updated, tile-by-tile; then the top row is sent back to its owner. In the original `ttmqr` code, updating each pair of rows was done in a single loop, as shown in Algorithm 1.

Algorithm 1 Original ttmqr algorithm

```
1     // Update rows i0, i1, each of k tiles.
2     // Rank src owns row i0, dst owns row i1.
3     for (j = 0, ..., k-1) {
4         if (tile (i0, j) is local) {
5             MPI_Send tile (i0, j) to dst
6             MPI_Recv tile (i0, j) from dst
7         }
8         else if (tile (i1, j) is local) {
9             MPI_Recv tile (i0, j) from src
10            tpmqrt update of tiles (i0, j) and (i1, j)
11            MPI_Send tile (i0, j) back to src
12        }
13    }
```

This was a conservative implementation done in a single thread to ensure MPI correctness. However, it ignores potential parallelism. We refactored this into 3 loops: first communicate all tiles, then update all tiles in parallel, then communicate all tiles back, as shown in Algorithm 2. This still ensures MPI correctness by issuing MPI calls sequentially, but applies updates in parallel. A similar update applies to the LQ algorithm. The SVD two-stage reduction to band algorithm benefits from the changes as it calls the QR and LQ updates. Since this change reduces time spend on the CPU, which makes the GPU idle, it has a large impact on the overall QR, LQ and GELS performance. Figures Figures 3.2 to 3.4 show the performance gain of QR, LQ, and GELS, respectively, due to these improvements in addition to the infrastructure improvements described in Chapter 2.

Algorithm 2 Updated ttmqr algorithm

```
    // Update rows i0, i1, each of k tiles.
    // Rank src owns row i0, dst owns row i1.

    // 1. Send tiles.
    for (j = 0, ..., k-1) {
        if (tile (i0, j) is local) {
            MPI_Send tile (i0, j) to dst
        }
        else if (tile (i1, j) is local) {
            MPI_Recv tile (i0, j) from src
        }
    }

    // 2. Update tiles, in parallel.
    for (j = 0, ..., k) {
        if (tile (i1, j) is local) {
            #pragma omp task
            {
                tpmqrt update of tiles (i0, j) and (i1, j)
            }
        }
    }
    #pragma omp task wait

    // 3. Send updated tiles back.
    for (j = 0, ..., k) {
        if (tile (i0, j) is local) {
            MPI_Recv tile (i0, j) from dst
        }
        else if (tile (i1, j) is local) {
            MPI_Send tile (i0, j) back to src
        }
    }
```

At the same time, the code was also refactored to merge the `side == Left` and `side == Right` codes, which differ only in swapping i and j indices.

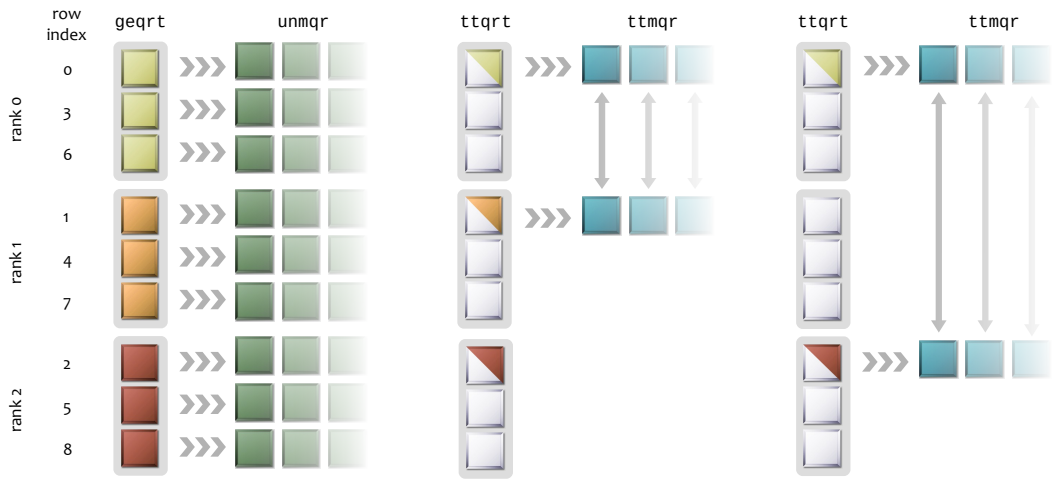

Figure 3.1

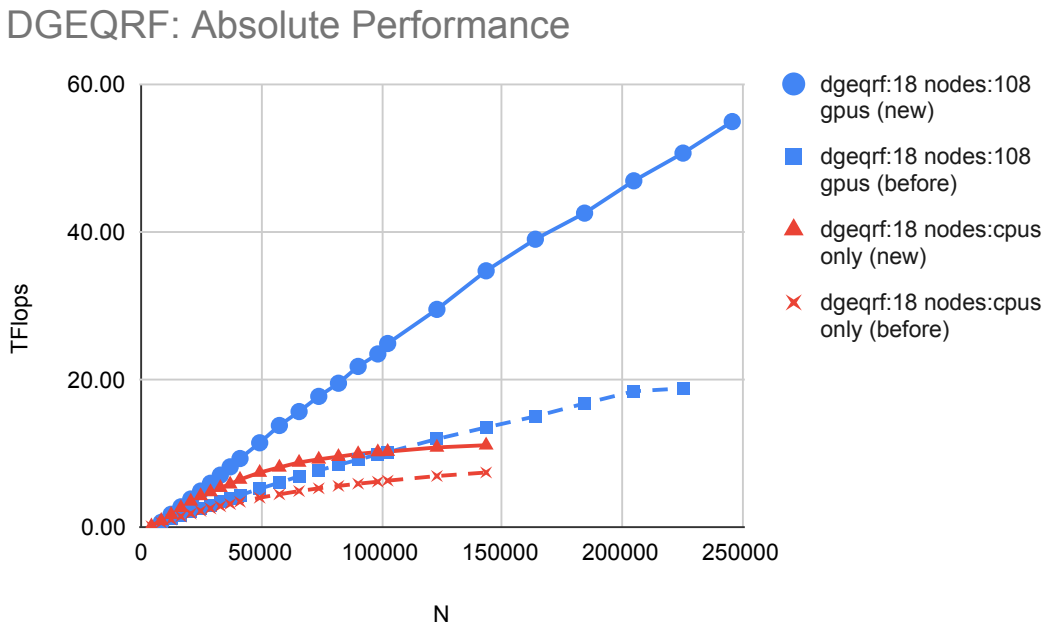

Figure 3.2: Performance of dgeqrf showing the effect of improvements for both GPU execution and CPU execution. Using 18 nodes on Summit, one process per socket, 6×6 process grid, tile size 192/256, inner blocking 16, panel threads 4.

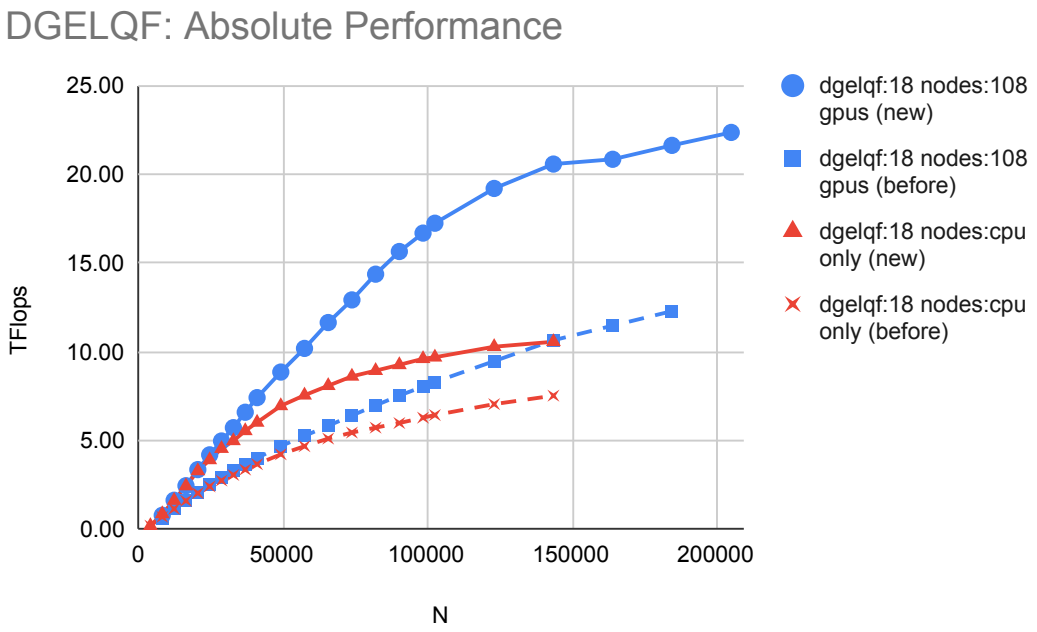

Figure 3.3: Performance of `dgelqf` showing the effect of improvements for both GPU execution and CPU execution. Using 18 nodes on Summit, one process per socket, 6×6 process grid, tile size 192/256, inner blocking 16, panel threads 4.

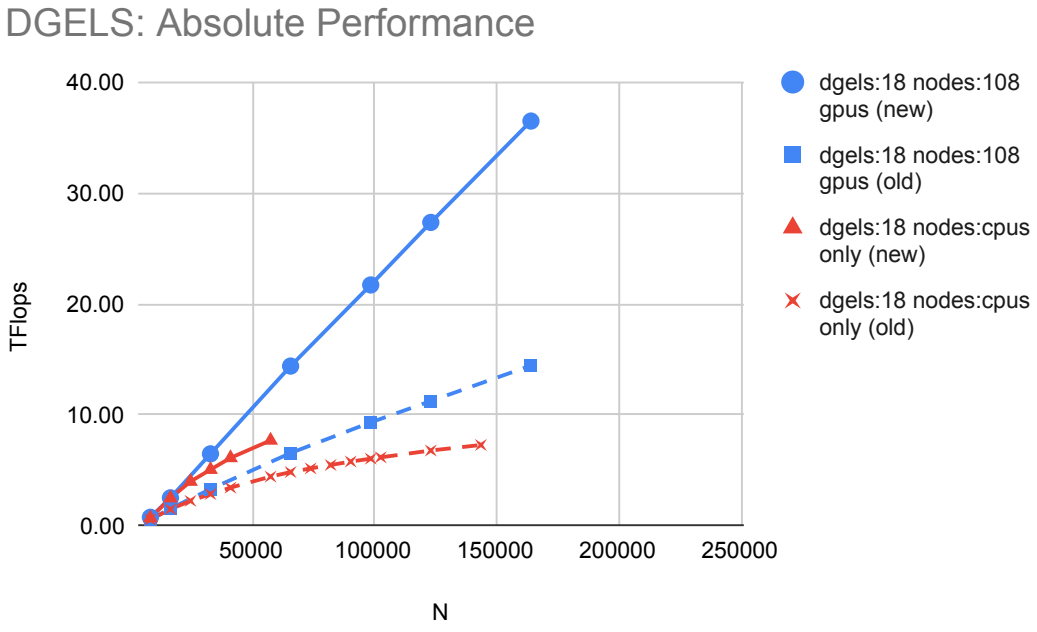

Figure 3.4: Performance of `dgels` showing the effect of improvements for both GPU execution and CPU execution. Using 18 nodes on Summit, one process per socket, 6×6 process grid, tile size 192/256, inner blocking 16, panel threads 4.

CHAPTER 4

Norm Improvements

This chapter describes the follow-up performance engineering techniques to enhance SLATE's multi-threaded performance for the one-norm and infinity-norm. The one-norm and infinity-norm have slightly higher complexity than the other types of norms due to the need to accumulate partial sums both along rows and columns.

Revisiting the partial sums accumulations at the computational routines level, a huge performance improvement has been achieved by avoiding the zero tiles corresponding to partial sums that do not reside on the node. Node locality is checked through the `A.tileIsLocal(i, j)` call. Another performance boost accomplished on the tile level by using a direct pointer to access the row elements of a tile (`A(i, j)`). Direct pointer access is accomplished by defining `scalar_t*Aj = &A.at(0, j)`, then using `Aj[i]` to access its row elements.

Figure 4.1 highlights the performance impact of various optimizations on the infinity norm using various number of nodes. It achieves gains up to [14.3x, 44.3x, 120x, 242x] using [2, 4, 8, 16] nodes, respectively. Also, Figure 4.2 reports SLATE's infinity norm against ScaLAPACK infinity norm. The SLATE's infinity norm achieves gains up to [15%, 29%, 31%, 20%] using [2, 4, 8, 16] nodes, respectively.

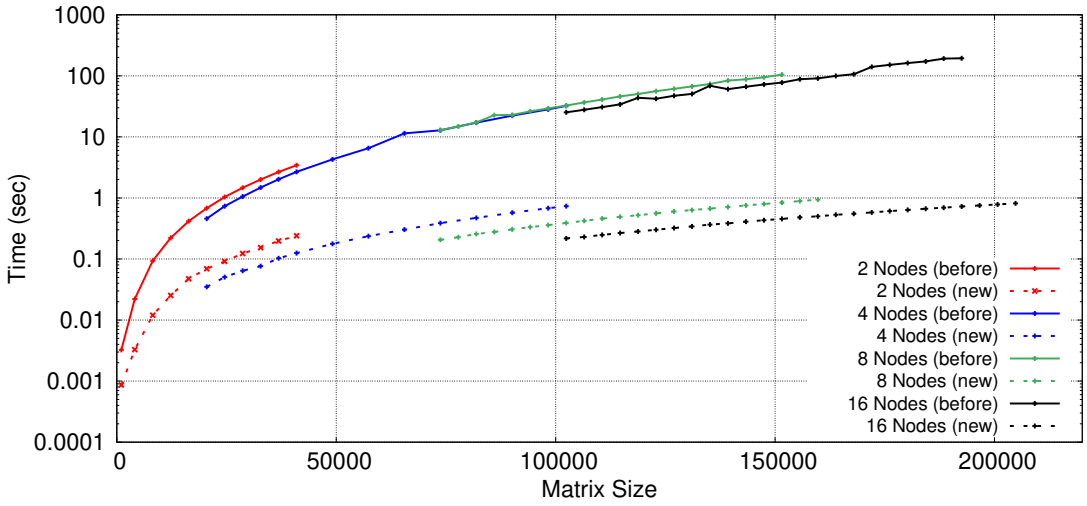

Figure 4.1: Performance of norm showing the effect of improvements for CPU execution only. Using 2, 4, 8 and 16 nodes on Summit, one process per socket, with 2×2, 2×4, 4×4 and 4×8 process grids, respectively, tile size 512.

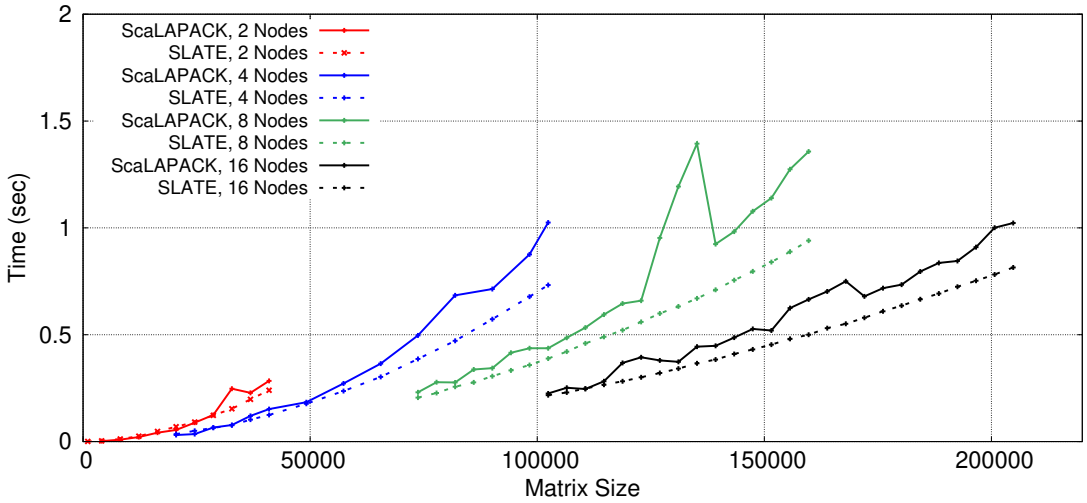

Figure 4.2: Performance comparison of SLATE and ScaLAPACK norm on CPU. Using 2, 4, 8 and 16 nodes on Summit. SLATE's tests using 2×2, 2×4, 4×4 and 4×8 process grids. ScaLAPACK's tests using 8×10, 10×16, 16×20 and 20×32 process grids.

CHAPTER 5

PBLAS Improvements

5.1 GEMM

For large C matrices, SLATE implements the general matrix multiplication (`slate::gemm`) as a sequence of block outer product `gemm` operations, the classical SUMMA algorithm. Each outer product `gemm` is implemented as a batch gemm operation on devices (`internal::gemm`). SLATE also uses its lookahead mechanism to overlap data transfer of (local and remote) tiles needed in the next iteration `internal::gemm` with the current iteration computations. To orchestrate this overlap, SLATE uses OpenMP tasks with dependencies.

On the other hand, issuing independent tasks that launch CUDA kernels on devices is an effective way to saturate the devices with enough computational load for better performance. However, launching multiple CUDA kernels and data transfer requests to the same driver is bottlenecked by the CUDA driver. Although executing these kernels on different CUDA streams gives way for possible overlap of computation and communication, the stage of launching these kernels is serialized at the CUDA driver calls. As such, the order by which these kernels are launched is critical to ensuring the overlap happens. Since there is no guarantee on the order by which the OpenMP tasks launching the CUDA kernels get executed, often the launching of the compute intensive CUDA kernels inside the `internal::gemm` get delayed by the launching of lookahead data transfers.

To overcome this scheduling hazard, we implement a two stage `internal::gemm`: an `internal::gemmPrep` that prepares the outer product gemm by fetching the tiles' data and collecting the corresponding pointer arrays, and another `internal::gemmExec` that executes the outer product gemm. OpenMP dependencies are used to guarantee correct execution order. Splitting the internal gemm into a fetch-task and an execute-task allowed a better pipelined

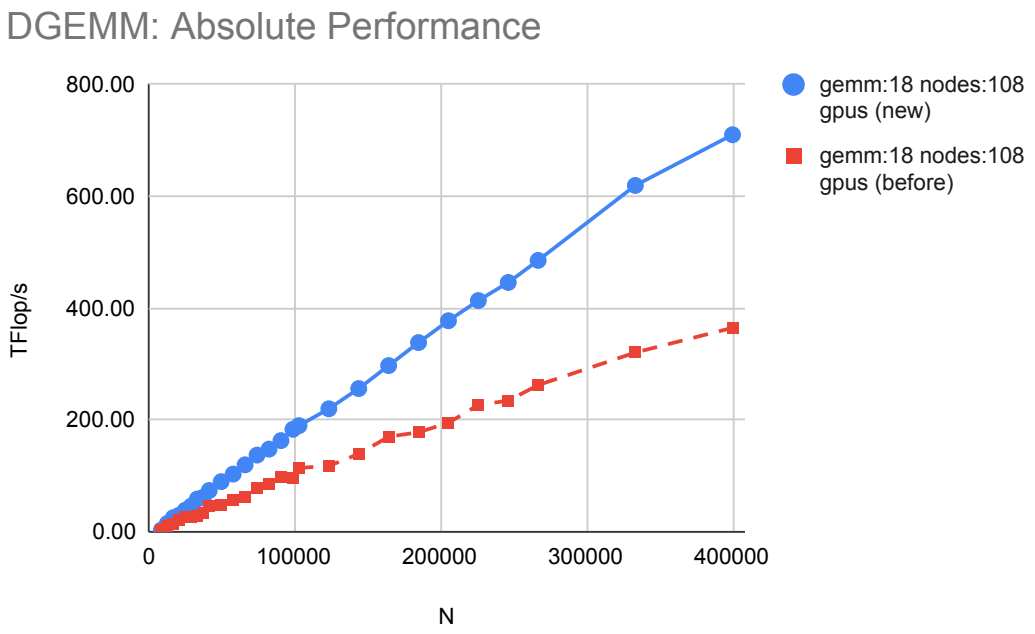

Figure 5.1: Performance of dgemm showing the effect of improvements for GPU execution. Using 18 nodes on Summit, one process per socket: 6×6 process grid.

execution of data transfer and kernel computations, and drastically improved the performance of the `slate::gemm` computational routine.

In addition, we enhance the performance of `slate::gemm` by early fetching the output matrix in parallel with MPI broadcasting of the first outer-product gemm. Figure 5.1 shows the cumulative overall performance gain of `slate::gemm` brought by the general infrastructure improvements as well as the split `internal::gemm` improvement.

5.2 HERK and SYRK

Similar to `slate::gemm`, the `slate::herk` computational routine is implemented as a sequence of outer-product `internal::herk` calls. An `internal::herk` routine is implemented as a batched-gemm on devices for the off-diagonal tiles, and a set of independent, parallel `herk` operations on the diagonal tiles executed on the host.

However, on the Summit supercomputer, there is a load imbalance between the host and the devices during the `slate::herk` computation, as the diagonal tiles herk operation on the host takes more time than the batch gemm for the off-diagonal tiles executed on the powerful NVIDIA V100 devices. To avoid the delays caused by such load-imbalance, we execute the diagonal tiles herk operations on devices using the NVIDIA cuBLAS library.

Figure 5.2 shows the performance improvement of `slate::herk` brought by the general infrastructure improvements as well as the device herk improvement.

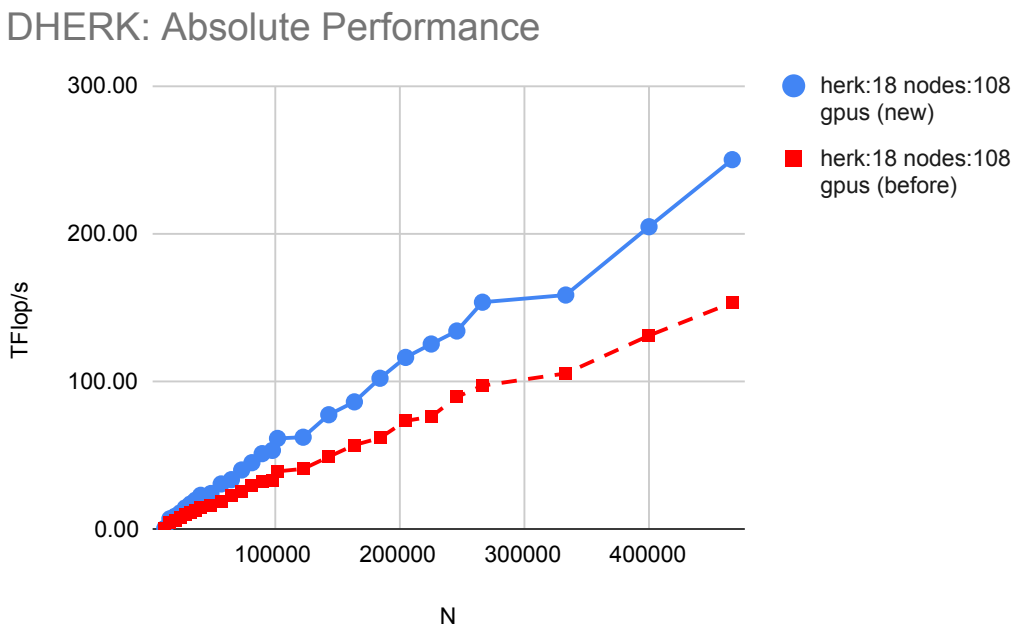

Figure 5.2: Performance of dherk showing the effect of improvements for GPU execution. Using 18 nodes on Summit, one process per socket: 6×6 process grid.

`slate::syrk` is the same as `slate::herk`, but for a symmetric instead of Hermitian matrix, and has the same considerations. In real arithmetic, they are identical.

5.3 HER2K and SYR2K

`slate::her2k` is structured very similar to the `slate::herk` computational routine. We enhanced the performance of `slate::her2k` by executing the diagonal tiles her2k on devices. Figure 5.3 shows the performance improvement of `slate::her2k` brought by the general infrastructure improvements as well as the device her2k improvement.

`slate::syr2k` is the same as `slate::her2k`, but for a symmetric instead of Hermitian matrix, and has the same considerations. In real arithmetic, they are identical.

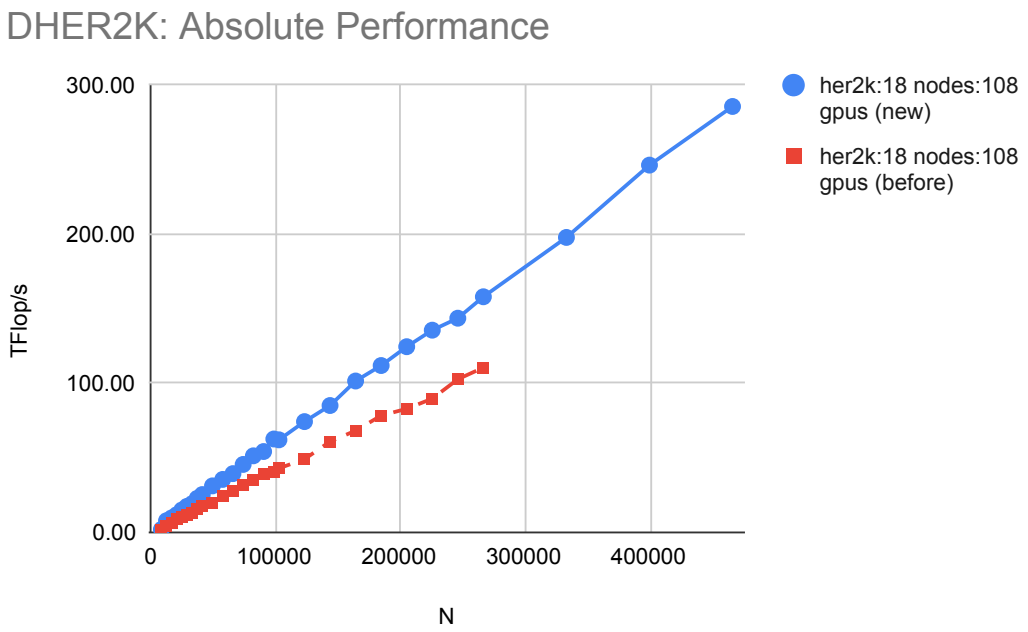

Figure 5.3: Performance of dher2k showing the effect of improvements for GPU execution. Using 18 nodes on Summit, one process per socket: 6×6 process grid.

CHAPTER 6

POTRF Improvements

SLATE implements the tile Cholesky decomposition algorithm. The original `Device` implementation executes the panel update tasks and the lookahead tasks on the host CPUs, and the GPUs carry out the trailing matrix update tasks. This division of tasks assumes that the target architectures contain nodes with the smaller CPU tasks reasonably balanced against the GPUs trailing matrix tasks. However, recent GPU architecture performance often surpasses the CPU counterpart to the degree that this division of tasks is no longer sustainable.

We optimized SLATE's Cholesky kernel by relying more on the performance obtained from the GPU devices. We first moved the panel update `trsm` and lookahead's `gemm` onto the GPU. In order to do this we implemented a SLATE interface to the batched `cublasXtrsmBatched()` routine. The diagonal herk is also moved to the GPU to avoid data transmission back-and-forth between the host-and-device throughout the kernel execution. Since NVIDIA does not provide a batch implementation of `herk`, we thereby call the basic CUDA `cublasXherk()` instead.

Executing the internal routines of SLATE's Cholesky factorization on the device alongside the `gemm` of the trailing matrix update causes two critical issues: data hazard and memory consistency. The data hazard happens when multiple routines operate on the same matrix object at the same time. Since SLATE allocates the GPU workspaces when the matrix object is allocated, any kernels that access the matrix object at the same time update these workspaces, which causes the data hazard in the form of Read After Write (RAW) and Write After Read (WAR) on these workspaces. We overcome this issue by implementing a mechanism to allocate multiple GPU workspaces in SLATE as needed.

The second issue is memory consistency, which arises because of a race condition between the `gemm` kernel call in the lookahead tasks and trailing matrix update tasks. Since the lookahead tasks are expected to finish before the trailing matrix update, the former one releases the matrix tiles before the latter one finishes. To overcome this issue, the broadcast call inside the

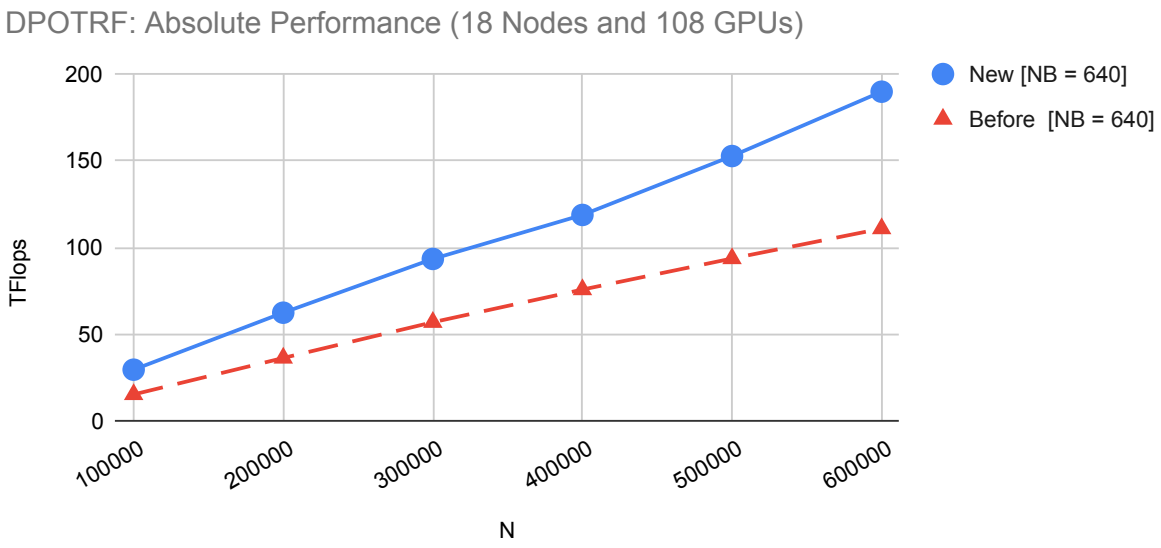

Figure 6.1: Performance of `dpotrf` showing the effect of improvements for hybrid CPU+GPU execution. Using 18 nodes on Summit, one process per socket: 6×6 process grid, and tile size: 640.

panel-update tasks copy these tiles into the target GPUs and hold them there. A new task is created after to release these tiles and free up the GPU memory.

Figures 6.1 and 6.2 show the performance improvement gains obtained by executing most of the Cholesky decomposition kernels on the GPU in addition to the infrastructure improvements described in Chapter 2. Note: In the optimized tests, we change the local GPU data distribution from "1D block column cyclic" to "1D block row cyclic" distribution to maximize the data locality across the devices and minimize data synchronization. Since most of the internal kernels of the Cholesky decomposition are now executed on GPU devices, we observe a substantial performance boost when "1D block row cyclic" distribution is used across the local GPUs, instead of "1D block column cyclic".

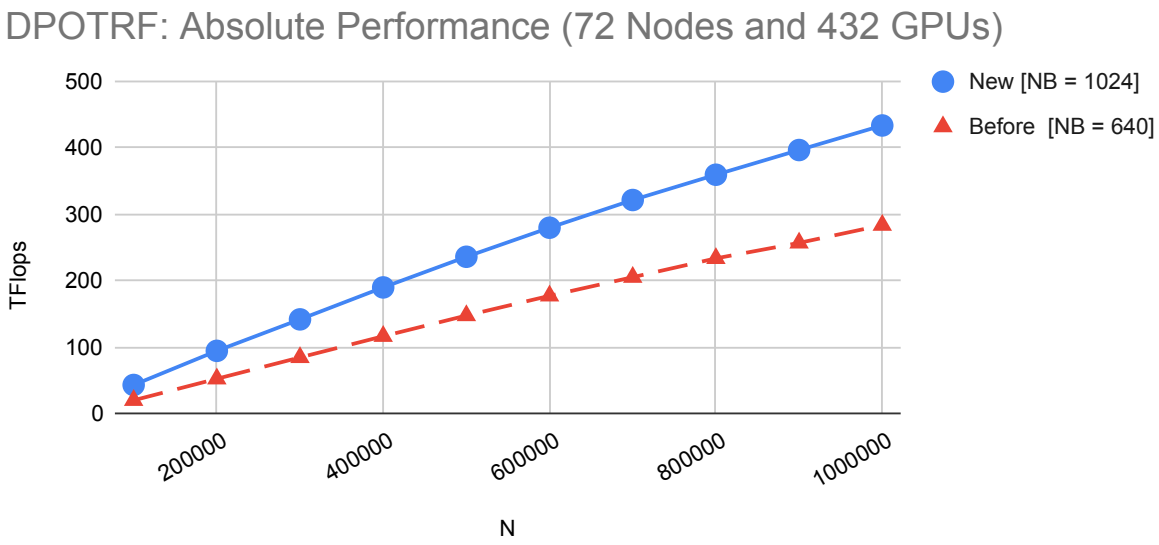

Figure 6.2: Performance of `dpotrf` showing the effect of improvements for hybrid CPU+GPU execution. Using 72 nodes on Summit, one process per socket: 12×12 process grid, and tile size: 640 (before) and 1024 (new).

Bibliography

[1] J. Hines. Stepping up to Summit. *Computing in Science Engineering*, 20(2):78–82, Mar 2018. doi: 10.1109/MCSE.2018.021651341.

[2] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, and Jack Dongarra. Least squares solvers for distributed-memory machines with gpu accelerators. In *Proceedings of the ACM International Conference on Supercomputing*, pages 117–126. ACM, 2019.