SLATE Performance Report: Updates to Cholesky and LU Factorizations

Asim YarKhan
Mohammed Al Farhan
Dalal Sukkari
Mark Gates
Jack Dongarra

Innovative Computing Laboratory

October 2, 2020

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**
10-2020	first publication

```
@techreport{yarkhan2020slateperformance,
  author={YarKhan, Asim and Al Farhan, Mohammed and and Sukkari, Dalal
   and Gates, Mark and Dongarra, Jack},
  title={{SLATE} Performance Report: Updates to {Cholesky} and
    {LU} Factorizations},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2020},
  month={Oct},
  note={revision 10-2020}
}
```

Contents

Contents ii

List of Figures iii

1 Introduction 1

2 Cholesky Performance 2
- 2.1 Tracing Cholesky . . . 2
- 2.2 Improved Batch TRSM under CUDA 11 . . . 2
- 2.3 Multithreading the tile broadcast operation . . . 3
- 2.4 Switching to CUDA-Aware MPI and GPUDirect . . . 3
- 2.5 Performance Improvements . . . 4

3 LU Performance 6
- 3.1 Profiling the LU Factorization . . . 6
- 3.2 Multi-threaded LU panel . . . 7
- 3.3 Gang scheduling of LU panel . . . 7
- 3.4 Moving additional tasks to GPU . . . 8

4 Conclusion 10

Bibliography 11

List of Figures

2.1 Improvement of batch-trsm from CUDA 10 to CUDA 11. The CUDA 10 figure on left shows that the batch-trsm operation took almost the same time as the trailing-matrix update operation. The CUDA 11 figure on the right shows the batch-trsm operation is now just a fraction of the trailing-matrix update operation. (Partial view of a Cholesky factorization trace on 2 Summit nodes using a single NVIDIA V100 GPU per node.) 3
2.2 Improved performance of the complete Cholesky factorization on 16 nodes of Summit (96 V100 GPUs) 4
2.3 Cholesky performance scaled by the number of NVIDIA V100 GPUs on 16 nodes of Summit (96 GPUs) 5

3.1 Partial trace focusing on a single GPU for a double-precision LU factorization of a large matrix using 16 nodes (96 NVIDIA V100s) on Summit. (Matrix size 64000, tile size 704, 6x16 process grid.) 6
3.2 Performance of the multi-threaded LU panel varying with the number of panel threads and MPI ranks. The matrix is only one panel (nb) wide, so the performance displayed is purely due to the multi-threaded panel. 7
3.3 Deadlock issue with multi-threaded tasks. 8
3.4 Performance improvement using HClib for gang scheduling in LU, compared to stock LLVM. 9

CHAPTER 1

Introduction

SLATE (Software for Linear Algebra Targeting Exascale) [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). The objective of SLATE is to provide fundamental dense linear algebra capabilities to the US Department of Energy and to the high-performance computing (HPC) community at large.

This report will discuss current efforts in improving performance in SLATE focusing on the Cholesky and LU factorizations. These improvements are intended to be general and many of them should be applicable to the other algorithms implemented in SLATE.

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

CHAPTER 2

Cholesky Performance

2.1 Tracing Cholesky

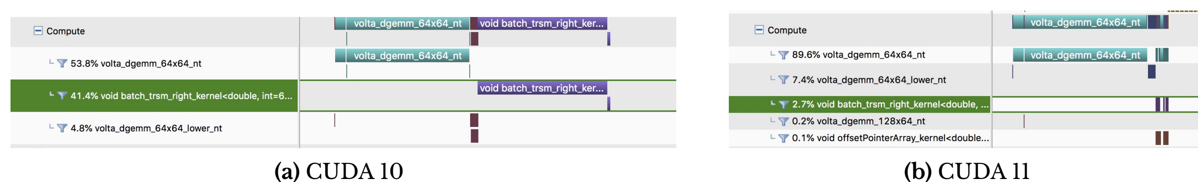
The performance of the Cholesky factorization was benchmarked on the Summit platform at ORNL. Both the performance and scalability were found to be unexpectedly lagging. The implementation was traced in order to discover the underlying causes of the lag in performance.

SLATE algorithms are designed to perform the large trailing-matrix update operations on GPUs/accelerators using batch operations (see the SLATE Developers' Guide for more details on the algorithmic implementations [1]). The panel and communication parts of the algorithm were originally designed to take place on the CPUs, with a lookahead factor allowing overlap between the trailing-matrix update operation communication and the panel/communications.

2.2 Improved Batch TRSM under CUDA 11

From traces it became evident that large batch trailing-matrix update operations on the accelerators were more efficient than expected, consuming very little time, making the panel operation on the CPU the bottleneck for the execution. For our Cholesky factorization the panel operation (batch-`trsm`) was then transferred to the GPU [2].

In CUDA versions prior to CUDA 11, the batch-`trsm` was slow relative to the batch-update operation so the performance improvement was less than expected. In CUDA 11, NVIDIA implemented a very fast batch-`trsm` and this operation was no longer part of the bottleneck. This improvement can be easily seen in the trace in Figure 2.1.



(a) CUDA 10 **(b)** CUDA 11

Figure 2.1: Improvement of batch-trsm from CUDA 10 to CUDA 11. The CUDA 10 figure on left shows that the batch-trsm operation took almost the same time as the trailing-matrix update operation. The CUDA 11 figure on the right shows the batch-trsm operation is now just a fraction of the trailing-matrix update operation. (Partial view of a Cholesky factorization trace on 2 Summit nodes using a single NVIDIA V100 GPU per node.)

CUDA 11 made the batch-`trsm` much faster relative to the batch-`gemm` matrix update.

2.3 Multithreading the tile broadcast operation

Traces from larger scale runs of the Cholesky factorization were examined to find further bottlenecks in the SLATE implementation. The MPI communication was found to be the next bottleneck.

Much of the communication in SLATE is structured as sets of multi-casts of individual tiles. An individual tile(i, j) is sent to the sub-matrices that require tile(i, j) for the computation. In the original code, this set of multi-casts is implemented using a sequential loop over all the items in the set. This sequential loop takes each tile(i, j) and handles the receives and sends (using an N-ary hypercube communication overlay). The MPI implementation was expected to use all the available bandwidth to achieve high performance.

Based on traces of Cholesky factorization on larger problems using more nodes, it was noted that the sequential loop was not sufficient to get the desired MPI performance. The loop was transformed using `omp taskloop` into a multi-threaded implementation (this requires `MPI_Thread_Multiple`). The new implementation uses each thread to work on a different tile(i, j) to do the receives and sends. This obtains a much larger performance for the communication for Cholesky factorization.

This change to multi-threaded communication has an impact on some routines that were using multiple threads to do panel factorization. The threads can be oversubscribed or otherwise in contention for access to communication hardware. These impacts are being currently evaluated before this change is made part of the default SLATE distribution.

2.4 Switching to CUDA-Aware MPI and GPUDirect

MPI communications in SLATE were originally designed to send messages from local CPU to the remote CPU, rather than directly using the accelerator. This decision was made because SLATE is intended to be easily portable, so there was an effort to avoid depending on specific

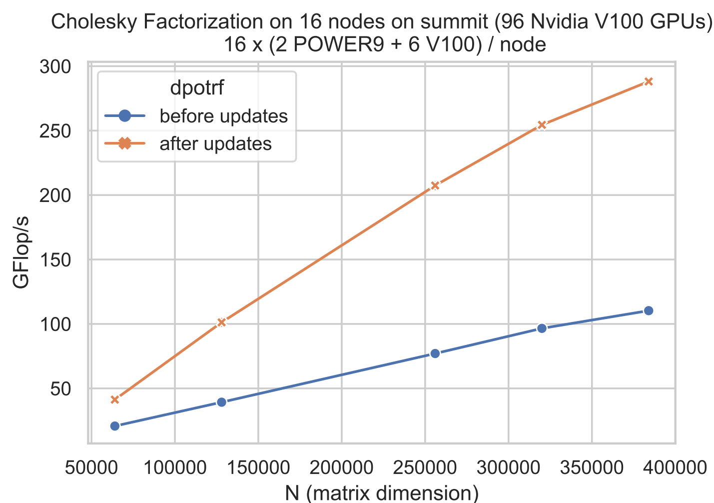


Figure 2.2: Improved performance of the complete Cholesky factorization on 16 nodes of Summit (96 V100 GPUs)

accelerator extensions. MPI now provides transparent access to GPU-direct communications, so SLATE is being transitioned to using CUDA-aware MPI that uses NVIDIA's GPUDirect.

Prior to this update, SLATE algorithms used CUDA memory copy operations to transfer data from the GPU to the CPU, do the communication from CPU-to-CPU, and then use CUDA memory copy to transfer data back to the GPU.

The Cholesky factorization does the majority of its communication via multi-casts of individual tiles. These MPI communication have been adapted to transfer data directly between GPU devices to improve the bandwidth.

Transferring data between GPUs has impacts that require additional work before this update is pushed into the default SLATE repository. The implementation needs to be able to smoothly switch between MPI implementations that are CUDA-aware and those that are not.

2.5 Performance Improvements

The effect of our performance updates can be seen in the performance of the double-precision Cholesky factorization shown in Figure 2.2 The original implementation before these updates achieved about 110 TFlop/s on 16 nodes of Summit (using 96 V100 GPUs) when factoring a matrix of size $N = 384000$ and the tile size $nb = 960$.

The updates described in this section were applied to Cholesky factorization, that is, using multi-threaded MPI messaging, switching the data locations to take allow CUDA-aware MPI, and changing to CUDA 11 to take advantage of the improved batch-`trsm`. On 16 nodes of Summit (96 V100 GPUs) the maximum performance reached was then 289 TFlop/s. Improved performance can be seen throughout the range for all matrix sizes.

The performance is also viewed when scaled by the number of V100 GPUs (i.e., scaled by

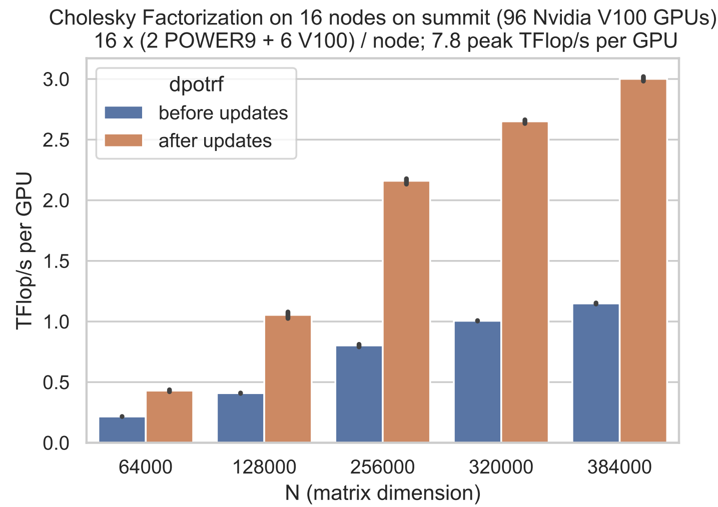


Figure 2.3: Cholesky performance scaled by the number of NVIDIA V100 GPUs on 16 nodes of Summit (96 GPUs)

96) in Figure 2.3. Figure 2.3. Each NVIDIA V100 GPU has a theoretical peak performance of 7.8 TFlop/s. The SLATE implementation went from achieving 1.1 TFlop/s (out of 7.8) before updates, to achieving 3.0 TFlop/s (out of 7.8 TFlop/s) after updates.

CHAPTER 3

LU Performance

3.1 Profiling the LU Factorization

The original version of the LU factorization was profiled using the NVIDIA profiling tools to determine the bottlenecks in the NVIDIA GPU implementation of the algorithm.

Since the focus is on keeping the GPU occupied, only the portion of the trace on the GPU is shown in Figure 3.1. In the figure the batch-`gemm` operation (the `volta_dgemmm` highlighted in green) is so efficient on the NVIDIA V100 that it takes a very small part of the time. The CPU parts of the algorithm (panel factorization and MPI communication) are a major bottleneck in the LU operation. Additionally, the swap operation which was implemented on the GPU is taking much longer than expected, it is being slowed down partially by a large number of host-device memory copy operations which occur as part of the swap operation.

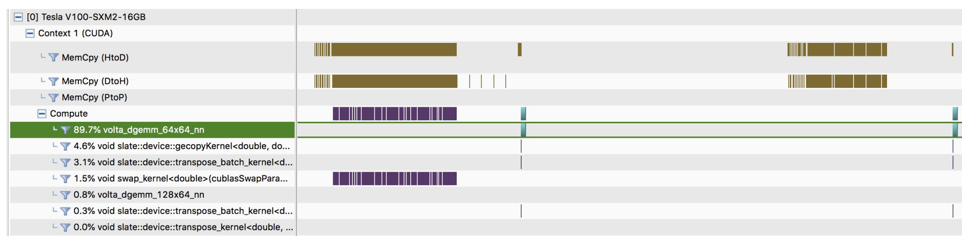


Figure 3.1: Partial trace focusing on a single GPU for a double-precision LU factorization of a large matrix using 16 nodes (96 NVIDIA V100s) on Summit. (Matrix size 64000, tile size 704, 6x16 process grid.)

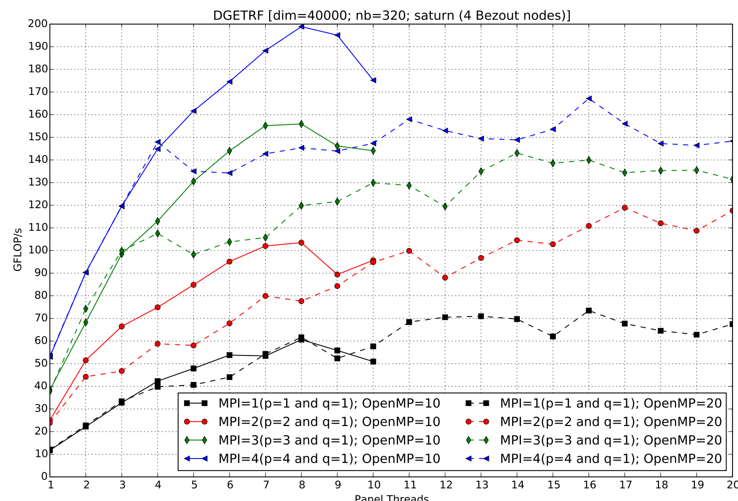


Figure 3.2: Performance of the multi-threaded LU panel varying with the number of panel threads and MPI ranks. The matrix is only one panel (nb) wide, so the performance displayed is purely due to the multi-threaded panel.

3.2 Multi-threaded LU panel

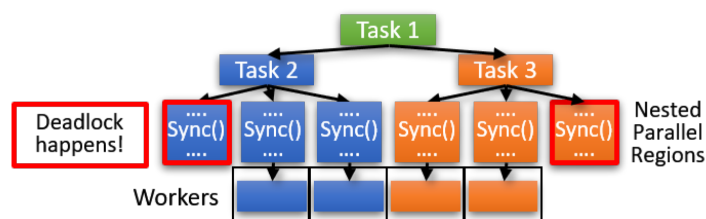
If the panel in the LU factorization is not fast, it can become a bottleneck in the entire computation, especially when paired with an accelerated trailing matrix update. From the beginning, the LU panel was designed to be multi-threaded on the CPU. We performed experiments to determine if this multi-threaded panel implementation was scaling with increasing numbers of threads assigned to the panel (panel threads).

In Figure 3.2 it is seen that the as the number of panel threads and MPI ranks increases, the performance increases, till some maximum performance. This implies that the multi-threaded panel is performing as expected when it is executed in isolation. There may be some additional effects when the panel is run as part of a complete LU factorization, where other threads may cause contention or over-subscription.

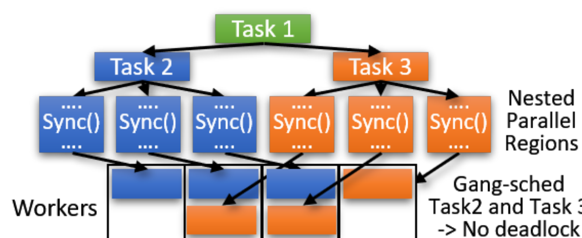
3.3 Gang scheduling of LU panel

While the panel is written as a multi-threaded task, OpenMP does not have the concept of a multi-threaded task. The panel can be started as several separate sub-tasks, or as a single task with nested parallelism inside. In either case, the sub-tasks in the panel are required to synchronize for each column of the panel to do a local max reduction to find the pivot. Additionally, an MPI reduction and row swapping occurs for each column of the panel. If all these sub-tasks assigned to the panel do not start simultaneously, there can be a significant synchronization delay or even deadlock, as shown in Figure 3.3.

To investigate this issue, we collaborated with Seonmyeong Bak, Oscar Hernandez, and Vivek Sarkar from the ECP SOLLVE team, who are working on enhancements to OpenMP for exascale machines in their Habanero-C threading library (HClib) with the LLVM OpenMP runtime.



(a) Deadlock in nested parallel tasks



(b) Deadlock avoidance with gang-scheduling of nested parallel tasks

Figure 3.3: Deadlock issue with multi-threaded tasks.

The SOLLVE team implemented a gang scheduling algorithm to start all the panel sub-tasks simultaneously, to avoid deadlock and synchronization delays. These exhibited modest performance improvements in Figure 3.4 using 1 node of the NERSC Cori GPU machine ($2\times$ 20 core Intel Skylake 6148, $8\times$ NVIDIA V100 GPU). A detailed discussion of this collaboration has been submitted to PPoPP'21 [3].

3.4 Moving additional tasks to GPU

SLATE implements LU factorization using OpenMP tasks for the panel factorization, lookahead update, and trailing matrix update, with data dependencies specified between the tasks. The original GPU implementation assigns some tasks to the CPU and other tasks to the GPU. The panel tasks and the lookahead tasks are executed on CPUs. The trailing-matrix update task contains both CPU and GPU parts: the CPU part does a `trsm` and the GPU part does a batch-`gemm` and internal row pivoting.

Based on the observations from our tracing, we started optimizing SLATE's LU kernel by moving more of the work to the GPU and trying to obtain more performance from the GPU devices. We moved all of the lookahead's internal kernels (row pivoting, batched `trsm`, and batched `gemm`) onto the GPU devices. Since row pivoting on GPUs is known to perform better when the data layout is in row-major [4], we implemented the row-major layout of the internal batched `trsm` kernel (invoking `cublasXtrsmBatched` with row-major layout). That way we avoid excessive tile transposition when moving from the row pivoting kernel to the other internal routines.

In addition, the MPI broadcast routine, which is invoked in the panel update task to broadcast

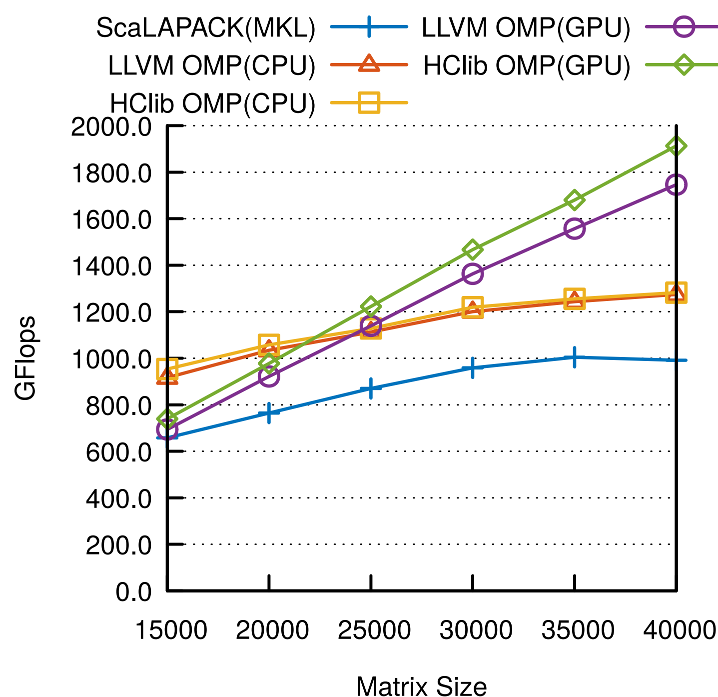


Figure 3.4: Performance improvement using HClib for gang scheduling in LU, compared to stock LLVM.

the panel column after the panel internal `getrf` and in the lookahead update and trailing-matrix update tasks to broadcast the updated rows after the internal batch-`trsm`, is updated to initialized the GPU devices with missing tiles (transposed to row-major).

Executing multiple internal routines of SLATE's LU factorization simultaneously on the GPU causes two critical issues: data hazards and race conditions. Each batched BLAS call takes batch arrays of tile pointers, `A_array`, `B_array`, and `C_array` for the `A`, `B`, and `C` matrices, respectively. These batch arrays are allocated in the matrix object. Originally, there was only one set of batch arrays, so executing simultaneous kernels would cause data hazards in the form of Read-After-Write (RAW) and Write-After-Read (WAR) on these arrays. We overcome this issue by allocating multiple GPU batch arrays as needed, one for each lookahead task and one for the trailing matrix task. The second issue is a race condition in releasing workspace tiles after internal batch-`gemm` calls. The lookahead tasks are expected to finish before the trailing matrix update, and would prematurely release workspace tiles that the trailing matrix update was still depending on. To overcome this issue, the broadcast call that copies tiles to the target GPUs also sets a hold on them, so they will not be released by the lookahead task. A new task is created after the trailing matrix update to release the hold on these tiles and free up the GPU memory.

CHAPTER 4

Conclusion

The implementation of the Cholesky and LU factorizations in SLATE were traced and examined. Bottlenecks were discovered and progress was made in addressing them. The performance of Cholesky implementation was substantially improved by 2.7× on GPUs and this was demonstrated on the Summit platform. The LU factorization is a tougher problem, and though some progress was made, the desired improvement was not yet achieved. Additional work progresses to move more of the kernels from the CPU to the GPU and to improve the swapping operation. Upcoming SLATE milestones will continue to focus on performance improvements.

The current snapshot of this performance update effort can be found in a development fork at https://bitbucket.org/icl/slate_devel. Please note that this is an active development fork and may not be stable.

Bibliography

[1] Ali Charara, Mark Gates, Jakub Kurzak, Asim YarKhan, Mohammed Al Farhan, Dalal Sukkari, and Jack Dongarra. SLATE developers' guide, SWAN no. 11. Technical Report ICL-UT-19-02, Innovative Computing Laboratory, University of Tennessee, December 2019. URL https://www.icl.utk.edu/publications/swan-011. revision 08-2020.

[2] Mark Gates, Ali Charara, Asim YarKhan, Dalal Sukkari, Mohammed Al Farhan, and Jack Dongarra. SLATE working note 14: Performance tuning SLATE. Technical Report ICL-UT-20-01, Innovative Computing Laboratory, University of Tennessee, January 2020. URL https://www.icl.utk.edu/publications/swan-014. revision 01-2020.

[3] Seonmyeong Bak, Oscar Hernandez, Mark Gates, Piotr Luszczek, and Vivek Sarkar. Task-graph scheduling extensions for efficient synchronization and communication. In *Principles and Practice of Parallel Programming 2021 (PPOPP'21) (submitted)*, 2021.

[4] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 157–160, 2014.