ADELUS: A Performance-Portable Dense LU Solver for Distributed-Memory Hardware-Accelerated Systems

Vinh Q. Dang, Joseph D. Kotulski, and Sivasankaran Rajamanickam

Sandia National Laboratories, Albuquerque, NM, 87123, USA
{vqdang,jdkotul,srajama}@sandia.gov

Abstract. Solving dense systems of linear equations is essential in applications encountered in physics, mathematics, and engineering. This paper describes our current efforts toward the development of the ADELUS package for current and next generation distributed, accelerator-based, high-performance computing platforms. The package solves dense linear systems using partial pivoting LU factorization on distributed-memory systems with CPUs/GPUs. The matrix is block-mapped onto distributed memory on CPUs/GPUs and is solved as if it was torus-wrapped for an optimal balance of computation and communication. A permutation operation is performed to restore the results so the torus-wrap distribution is transparent to the user. This package targets performance portability by leveraging the abstractions provided in the Kokkos and Kokkos Kernels libraries. Comparison of the performance gains versus the state-of-the-art SLATE and DPLASMA GESV functionalities on the Summit supercomputer are provided. Preliminary performance results from large-scale electromagnetic simulations using ADELUS are also presented. The solver achieves 7.7 Petaflops on 7600 GPUs of the Sierra supercomputer translating to 16.9% efficiency.

Keywords: Dense linear systems of equations · Distributed computing · GPU acceleration · LU factorization · Performance portability.

1 Introduction

Solving a dense linear equations system is one of the most fundamental problems in numerous applications in the mathematical sciences and engineering, such as biology [1], economics [2], electrical network analysis, aircraft design, radar technology [3], etc. We can find dense linear systems of equations in many applications involving the solutions of linear partial differential equations formulated as boundary integral equations (a.k.a. boundary element method) including acoustics, electrochemistry, fluid mechanics [4], elastodynamics, fracture mechanics [5], electromagnetics (method of moments) [6]. In these applications, the boundaries of the objects of interest are discretized and the integral equations are formulated into the form of **A*****x**=**b** where **A** is a dense, square matrix,

b is (are) the corresponding right-hand-side (RHS) vector(s), and **x** is (are) the unknown solution vector(s).

In order to solve **A*****x**=**b**, one typically uses direct solvers with lower-upper (LU) factorization, which decomposes the matrix **A** into a lower triangular matrix **L** and an upper triangular matrix **U** such that **A**=**L*****U**, due to its high accuracy and robustness. However, dense LU factorization has a high computational complexity of $O(N^3)$, and a memory requirement of $O(N^2)$ which might prevent itself from simulations of extremely large problems. To reduce the heavy computational burden of direct solvers, one can use iterative solvers with their computational complexities of $O(N^2\sqrt{\kappa})$ where κ is the condition number of matrix **A** [7]. Many efforts have also been devoted to further accelerate the iterative solvers. For instance, in the area of method of moments, many fast factorization schemes have been proposed in the literature to reduce the cost of matrix-vector multiplications in iterative solutions using some suitable expansions of the underlying integral kernel with some sacrifices of accuracy. Two well-known techniques are the fast multiple method (FMM) [8] and the multilevel fast multipole algorithm (MLFMA) [9] which can reduce the computational complexity to $O(N^{1.5}\sqrt{\kappa})$ and $O(N\log(N)\sqrt{\kappa})$, respectively.

Despite its high computational complexity, a direct solver often provides more robust results in cases where many iterative solvers fail to solve accurately and/or fail to converge because the system matrices are extremely ill-conditioned. Such problems, e.g. structures supporting high-quality factor resonances or extremely large problems compared to the wavelength, are very common in real-world applications. Therefore, it is essential to have efficient implementations of dense direct solvers. The dense formulation of the problem is also memory-intensive. Most problems of interest require several hundreds or thousands of nodes/GPUs to be able to fit in memory. Therefore the dense direct solvers have to be distributed-memory parallel as well. Dense LU factorizations are also compute-intensive algorithms ($O(N^3)$ FLOPS). Hence the distributed-memory, dense LU factorization has to be able to utilize the hardware accelerators available on several of the top supercomputers extremely well. This could mitigate their aforementioned computational cost and allow them to target extremely large-scale problems while providing robust solutions to applications. Some problems of interest to us such as the boundary element method applied to electromagnetics in the frequency domain [6] result in matrices **A** that are dense complex. Hence we need to support accelerator-focused, distributed, dense LU factorizations that can handle real and complex matrices. This is a challenging problem by itself. The challenge is made even harder by the diversity in the accelerator architectures.

The current second fastest machine on the TOP500 list is the Summit system [10] located at the Oak Ridge National Laboratory (ORNL). Each compute node of the Summit system has two POWER9 CPUs and six NVIDIA V100 GPUs. The peak double-precision floating-point performance of the CPUs and the GPUs per compute node are 1.08 TFLOPS and 46.8 TFLOPS, respectively. The third fastest supercomputer is the Sierra system [11] located at the Lawrence

Livermore National Laboratory (LLNL). Each node of Sierra has two POWER9 CPUs at 1.08 TFLOPS, and four V100 GPUs for 31.2 TFLOPS. We present performance results on both these systems (Section 6).

We also highlight three architectures that are of interest to us in the near future. Recently, the U.S. Department of Energy has announced plans for three exascale-class supercomputers: (1) Aurora system [12], at the Argonne National Laboratory, will be delivered in 2021 with sustained performance of 1 ExaFLOPS. Each Aurora node will contain two Intel Xeon scalable processors and six X^{e} architecture-based GPUs; (2) Frontier system [13] at the ORNL. It will be delivered in 2021 with 1.5 ExaFLOPS of theoretical peak performance. Each Frontier node will contain one AMD EPYC CPU and four purpose-built AMD Radeon Instinct GPUs; (3) El Capitan system [14] at the LLNL is scheduled for early 2023 with 2 ExaFLOPS of theoretical peak performance. Each El Capitan node will contain one AMD EPYC CPU and four next-generation AMD Radeon Instinct GPUs. These next generation exascale HPC architectures are continuously evolving to allow for solving larger, more computationally intensive problems. At the same time, they have introduced new challenges to algorithm designs and implementations due to significantly different architectures and programming models. Therefore, it is important to develop the dense LU solver based on algorithms and implementations that are portable to future platforms.

This paper presents ADELUS, a performance-portable dense LU solver for current and next generation distributed-memory hardware-accelerated HPC platforms. ADELUS computes the LU factorization with partial pivoting and solves real/complex dense linear systems in distributed-memory using the message passing interface (MPI). The matrix is block-mapped onto the MPI tasks (either stored on CPU memory or GPU memory). In this work, the torus-wrap mapping scheme [15], which is transparent to the users, was adopted for an optimal balance of computation and communication. MPI processes compute the factorization and solve the portion of the linear system as if the matrix was torus-wrapped. A permutation operation is performed to restore the results when the solve completes. In this work, we provide performance portability by leveraging the abstractions provided in the Kokkos programming model [16] and Kokkos Kernels library [17].

The main contributions of this paper are the following:

- A parallel, dense, performance-portable, LU factorization algorithm based on torus-wrap mapping.
- An implementation of the real/complex LU factorization algorithm for traditional and accelerator-based architectures that can achieve 1.397 PFLOPS on 900 GPUs on the Summit (the world's second fastest) supercomputer. The ADELUS software is available at https://github.com/trilinos/Trilinos.
- Comprehensive analysis of the performance, scalability, and the effect of using different memory spaces on distributed-memory.
- Integration of the dense LU solver into an electromagnetic application and a demonstration of application performance on 7600 GPUs with 7.720 PFLOPS on the Sierra (the world's third fastest) supercomputer.

2 Related Work

Dense LU factorization has been studied for several decades. In this section, we list the most popular software packages which implement LU solvers related to distributed memory and/or GPU accelerators. These algorithms and implementations are the most relevant with respect to our work. Distributed-memory LU factorization implementations are available in:

- ScaLAPACK [18]: ScaLAPACK is the standard library for high-performance dense linear algebra routines on distributed-memory computers. ScaLAPACK leverages BLAS and BLACS (Linear Algebra Communication Subprograms) for extending LAPACK routines to distributed-memory computing. The library is currently written in Fortran;
- Elemental [19]: Elemental is a C++ library for distributed-memory, dense and sparse-direct linear algebra, using C++ templates for multiple precision support. It interestingly distributes the matrix by elements, which is similar to the torus-wrap mapping scheme used in ADELUS. Since 2016, the Elemental library was forked by the LLNL team under the name Hydrogen, to make use of GPU accelerators. But the supported functionality is only limited to the basic utilities and BLAS-1,-3 operations;
- DPLASMA [20]: the DPLASMA library relies on the PaRSEC [21] runtime to schedule tasks from task dependency graphs, allowing for overlapping of communication and computation. DPLASMA, however, does not support either GPU acceleration for LU solver or C++ templates.

On the other hand, node-level hardware-accelerated implementations of the LU solvers are available in:

- CULA [22]: CULA Dense is a GPU-accelerated implementation of dense linear algebra routines providing a wide set of LAPACK and BLAS capability;
- MAGMA [23]: The MAGMA library aims to provide LAPACK functionalities for heterogeneous/hybrid architectures;
- cuSOLVER [24]: The cuSOLVER library is a high-level package based on the cuBLAS and cuSPARSE libraries. It provides useful LAPACK-like features, such as dense matrix factorization and solve routines such as LU, QR, etc.

The SLATE library [25] is the state-of-the-art library that targets multi-GPU-accelerated distributed-memory systems. SLATE provides coverage of existing ScaLAPACK functionalities, both accelerated CPU-GPU based and CPU based. SLATE uses a modern C++ framework with communication-avoiding algorithms, lookahead panels to overlap communication and computation, and task-based scheduling. To the best of our knowledge, ADELUS is the first effort addressing performance portability for LU solver via Kokkos/Kokkos Kernels libraries on distributed-memory accelerator-based architectures. We compare ADELUS' performance against some of these implementations in Section 6.

3 Overview of Kokkos and Kokkos Kernels

As the systems with several different accelerators become common, the need for portable programming model and portable algorithms has become critical. Portability can be addressed using several different approaches such as a directive-based approach (using OpenMP [26], OpenACC [27]), a library-based approach (using Kokkos [16], RAJA [28]) or by writing portable domain-specific languages (DSLs) if the target domain is small. Each one of these approaches has their advantages and disadvantages. In this work, we focus on the Kokkos performance-portable library to develop the dense LU solver. The primary reason we choose the library-based portable approach is due to the ability of this option to be used immediately with CPUs and GPUs effectively, and the availability of an ecosystem where options to call BLAS or LAPACK functionality is available through the Kokkos Kernels library [17].

Kokkos is a templated C++ library that uses meta-programming so users of the library will write the code once in templated C++. At compile time, these codes are mapped to an appropriate backend depending on compile time template parameters. There are backends available for OpenMP, CUDA for NVIDIA GPUs, and experimental backends for HIP for AMD GPUs, and SYCL for Intel GPUs. We use the OpenMP and CUDA backends in this work. Kokkos uses an *execution space* to determine where the computation is mapped and a *memory space* to determine where data structures live. Both aspects are key to performance. A **Kokkos View** is a data structure to store multidimensional arrays with reference counting. We utilize the Kokkos Views for storing the matrices and vectors. The matrices and vectors use different layouts depending on whether the data structures live on the CPUs or GPUs. In Kokkos library this is called **HostSpace** and **CudaSpace**. Furthermore, we also use **CudaHostPinnedSpace** for MPI buffers for better performance. Switching the data structures from one memory space to another is controlled completely at compile time with template parameters. The solver code remains the same for all the options.

Once the data structures are in place and an execution space is chosen, the key requirement for a dense linear solver is the availability of BLAS and LAPACK functionality. Kokkos Kernels library [17] provides portable sparse/dense linear algebra and graph kernels. It is implemented using Kokkos for portability. Kokkos Kernels also has interfaces to vendor-optimized BLAS/LAPACK when appropriate. There are custom BLAS/LAPACK kernels implemented for performance or functionality reasons as well. We depend on the Kokkos Kernels library for BLAS and LAPACK functionality on CPUs and GPUs. Kokkos Kernels uses the dense matrices stored in layouts optimized for CPU/GPU architecture and provides the BLAS/LAPACK functionality needed by the solver.

4 Application: Method of Moments for Linear Electromagnetics

An important class of problems that can be solved with the ADELUS solver are those encountered in the solution of the boundary element method applied to electromagnetics in the frequency domain. This class of problems solves Maxwell's equations in integral form by using the equivalence principle and employing divergence conforming basis functions for the currents on the *surfaces* of interest [6], [29]. In the electromagnetic's community, this is termed the method of moments. The matrix produced by this numerical technique is then solved by using ADELUS. Depending on how the boundary condition is applied, it can be categorized into two main approaches: (i) Electric Field Integral Equation (EFIE) where the boundary condition is applied on the electric field; (ii) Magnetic Field Integral Equation (MFIE) where the boundary condition is applied on the magnetic field. The EFIE can be applied to both open and closed objects whereas the MFIE applies only to closed objects. Without loss of generality, we provide a brief of summary of the method of moments for EFIE in this section.

Consider the equation on the surface S: $\mathcal{L}(J) = E$, where $\mathcal{L}$ is the linear operator derived from the EFIE, J is the unknown induced surface current, and E is the corresponding right hand side related to the incident field. $\mathcal{L}$ contains kernels in the form of Green's function $G(r, r') = e^{-jk|r-r'|}/|r-r'|$, where r and r' are an observation point on S and a source point on S, respectively, and k is the wave number. Let the current on S be approximated in terms of a basis function f_n defined on the surface as

$$J \cong \sum_{n=1}^{N} I_n f_n. \tag{1}$$

Typically, a triangular disretization of the surface is employed and the well-known Rao-Wilton-Glisson (RWG) function [29] is used as basis functions in (1). Applying (1) to the EFIE $\mathcal{L}(J) = E$ and using the Galerkin method to test each side of the equation yield a complex, dense, double-precision linear system

$$\sum_{n=1}^{N} \langle f_m, \mathcal{L} f_n \rangle I_n = \langle f_m, E \rangle, \tag{2}$$

where $m = 1, 2, ..., N$. Equation (2) has the form of $\mathbf{A} * \mathbf{x} = \mathbf{b}$ which can be solved by ADELUS for $\{I_n\}_{n=1}^{N}$. Note that the discretization required to solve problems of interest forces the usage of capability machines that are efficient in both message passing (MPI) and threading on advanced architectures (GPUs).

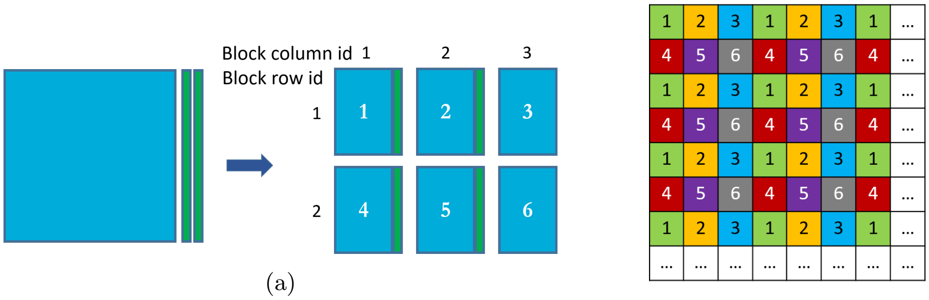
To this end, ADELUS has been successfully integrated with the method of moments code EIGER [30]. This production Fortran code has been used effectively for a large class of problems and on a variety of compute platforms – its utility has been extended by the ADELUS solver. The next generation version of EIGER, GEMMA [31], is currently being developed to use the Kokkos library to increase performance in the filling of the matrix as well.

5 Parallel LU Solver Implementation

In this section, we describe the implementation of ADELUS, including the matrix implementation using Kokkos, the torus-wrap mapping scheme, and the parallel LU solver using torus-wrap mapping (factorization, backward solve and permutation).

5.1 ADELUS Interface and Storage

ADELUS accepts a dense matrix and vectors that are block-mapped to the MPI processes. The matrix is distributed to the MPI processes such that the maximum difference in the number of rows (or columns) assigned to each MPI processes is at most one. The same rule is applied to the right hand side (RHS) vectors. ADELUS provides a distribution utility function for users to calculate the workload on each MPI process based on the number of columns (rows) of the matrix, the number of the RHS vectors and the number of processes assigned to a matrix row. The function returns the number of rows, columns and RHS vectors assigned to the process, the row and column addresses of the matrix portion in the global matrix, and the row and column indices of the matrix portion in the local block map. Fig. 1a shows an example of mapping the original matrix and two RHS vectors to six MPI processes with three processes per row. This utility function is used by our applications to assemble the portions of the matrix and the RHS vectors in the 2D block format correctly on each MPI process and provide them as input to ADELUS. ADELUS is then called by MPI processes taking the portions of matrix packed with RHS vectors as their inputs.



(a)

1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
...	...	...	...	...	...	...	...

(b)

Fig. 1: ADELUS workload distribution and torus-wrap mapping for 6 MPI processes (3 processes in a row), and 2 RHS vectors. The MPI process indices are shown in the boxes: (a) Workload distribution; (b) Torus-wrap mapping.

Similar to traditional dense linear solver packages, ADELUS stores its data (matrix and RHS vector portions) in each MPI process contiguously in the column-major order. For portability, the ADELUS data container is implemented by the Kokkos View with layout as Kokkos::LayoutLeft. Kokkos::LayoutLeft essentially forces Kokkos to use column major order. The Kokkos Views are allocated either in the host memory (**HostSpace**) or in the device memory (**CudaSpace**) depending on the desired execution backend (i.e. CPU, GPU, etc.). We use Kokkos::complex so the matrices and vectors remain portable on

CPUs and GPUs. For example, one can allocate a 2D view (matrix) of complex values in the host memory by:

```
Kokkos::View<Kokkos::complex<double>**,
             Kokkos::LayoutLeft,
             Kokkos::HostSpace>
             A(”A”,my_rows,my_cols);
```

or in the CUDA device memory by:

```
Kokkos::View<Kokkos::complex<double>**,
             Kokkos::LayoutLeft,
             Kokkos::CudaSpace>
             A(”A”,my_rows,my_cols);
```

Note that the constructor takes a string that is primarily used for debugging and profiling purposes. The *my_rows* and *my_cols* are local number of rows and columns in each MPI rank. The current ADELUS solver requires the matrix and RHS vectors are packed together and computed before ADELUS is called since the forward solve is integrated with the factorization of the matrix with the RHS appended next to the matrix. This scenario is very common in the computational electromagnetics where users usually compute the matrix and the RHS vectors before calling the solvers. In order to comply with other LU solvers, we are going to provide the GETRF and GETRS functionalities separately in the upcoming ADELUS versions.

In the current version of ADELUS, the implementation is exclusive to one architecture, that is, the matrix resides in either host memory (if running on CPU backend) or device (CUDA) memory (if running on GPU backend). We plan to target a hybrid implementation where host memory and device memory are both utilized in the future versions.

5.2 Torus-Wrap Mapping

The torus-wrap mapping scheme [15] is adopted for workload distribution in ADELUS. The advantages of this mapping are each process has nearly the same workload and the process idle time is minimized. Assuming the number of MPI processes P can be factored as $P = P_r \times P_c$, where P_r is the number of processes per column and P_c is the number of processes per row, one can construct a block mapping with the block sizes of $M_p \times N_p$, where M_p=N/P_r and N_p=N/P_c. If N is not divisible by P_r or P_c, some processes will be assigned one more row and/or column than others. Internally, ADELUS, which uses the torus-wrap mapping scheme, assigns columns 1, P_c+1, 2P_c+1, ... to processes 1, P_c+1, 2P_c+1, ...; columns 2, P_c+2, 2P_c+2, ... to processes 2, P_c+2, 2P_c+2, ... For rows, ADELUS assigns rows 1, P_r+1, 2P_r+1, ... to processes 1, 2, ..., P_c; rows 2, P_r+2, 2P_r+2, ... to processes P_c+1, P_c+2, ... In other words, the column indices assigned to a MPI process constitute a linear sequence with step size P_c, and the row indices are in a sequence separated by P_r. It is not necessary to redistribute the block-mapped matrix among processes for torus-wrapped solver [15]. More specifically, a block-mapped system can be solved by a solver assuming a torus-wrapped system. In ADELUS, the solution vectors are corrected afterwards by straightforward permutations. The details are transparent to the users. Fig. 1b shows an example of matrix elements torus-wrap mapped to 6 MPI processes with 3 processes per row. It should be noted that the performance of ADELUS depends on the distribution of matrix on MPI processes (i.e. the selection of P_c and P_r). It is common to choose $P_c \geqslant P_r$ for better performance. More detailed discussion is given in Section 6.3.

5.3 LU Solver

In ADELUS, the LU solver comprises three main steps: LU factorization+forward solve, backward solve, and permutation. We detail the algorithms for these steps in this section.

LU Factorization and Forward Solve As the forward solve is similar to the LU factorization in terms of data use/reuse, we merge the forward solve with the factorization for performance and coding simplicity. We implement the right-looking variant of the LU factorization with partial pivoting of a dense $N \times N$ matrix. The algorithm is summarized in Algorithm 1. Each iteration in Algorithm 1 has 4 steps:

- Step 1 is to *find* the pivot. An MPI column sub-communicator is formed for the processes that own column j. Each process *finds* its own local maximum entry in the column and then exchanges within the sub-communicator for the global pivot value.
- Step 2 is to *scale* the current column j of Z with the pivot value and generate column update vector from the column j. The pivot row index and the column update vector are communicated to processes sharing the same row sets.
- Step 3 is to exchange pivot row and diagonal row. The pivot row is first *updated* and then broadcasted within each column sub-communicator. The row owner processes also send the diagonal row to processors owning the pivot row.
- Step 4 is to *update* the current column, and if saving enough columns, to *update* Z via the outer product.

Each MPI process handles its own local matrices while using Kokkos Kernels BLAS interfaces which are implemented in a simple, generic way so that the resulting code is able to run on a wide range of architectures. The BLAS interfaces enable convenient calls to vendor library BLAS routines well-optimized for multi-threaded CPU and massively parallel GPU architectures. In this work, Kokkos Kernels calls IBM's ESSL BLAS when called with the CPU backend and calls cuBLAS when called with the CUDA backend. There are some exceptions where Kokkos Kernels calls its own implementations but they do not get used for our experiments in this work. Depending on where the data resides, Kokkos Kernels calls the right BLAS routines for the targeted backend. The BLAS operations needed in ADELUS include: (i) *KokkosBlas::iamax* for finding the local pivot entry in a column (Line 5 of Algorithm 1), (ii) *KokkosBlas::scal* for scaling the column with the inverse of pivot value (Line 10), (iii) *KokkosBlas::copy* for copying back and forth between the matrix and temporary containers (Lines 15, 20, 23, 25, 27, 31, and 33), (iv) *KokkosBlas::gemm* for updating the matrix (Lines 22, 38, and 40). Our algorithm requires only simple communication patterns consisting of point-to-point communication: MPI_Send, MPI_Recv, MPI_Irecv (Lines 16, 18, 29, 31, and 33 of Algorithm 1) and collective communication: MPI_Bcast, MPI_Allreduce (Lines 7 and 24). Furthermore, CUDA-aware MPI is exploited on GPU architectures which allows direct communication among GPUs without the need of buffering GPU data through host memory. ADELUS also has the option of using host pinned memory to buffer GPU data before communication which can be used for computer systems not having a high performance implementation of CUDA-aware MPI.

We employ the delay-updating technique (Line 39 of Algorithm 1) to take advantage of the better efficiency of level-3 BLAS *gemm* as compared to level-1 and level-2 BLAS operations. An appropriate block size parameter BLKSZ can help enhance the solver performance. A typical value of BLKSZ for CPU backend is 96 while a typical value of BLKSZ for GPU backend is 128. We determine these using several evaluations for different matrices. These numbers are used in our performance evaluation in Section 6. The algorithm utilizes an overlapping technique which performs column updates within a block one column at a time (Line 38). To minimize the waiting time, the algorithm attempts to do row work while waiting for a column to arrive (Line 35).

Backward Solve In this phase, the elimination of the RHS is performed by the process owning the current column using the Kokkos $parallel\_for$ (Line 4 through Line 6 of Algorithm 2). The results from the elimination step are broadcasted to all the processes within the MPI column sub-communicator (Line 7). The *KokkosBlas::gemm* is then called to update the RHS (Line 8). To prepare for the next iteration, the newly-computed RHS vectors are sent to the processes to the left.

Permutation Since the torus-wrap mapping scheme is assumed by the solver while the input matrix is not torus-wrapped, a permutation of the solution vectors must be carried out to "unwrap the results". The algorithm is quite straightforward. Each process that owns local solution vectors creates a temporary buffer for global solution vectors. The permutation simply involves Kokkos $parallel_for$s to fill the local vectors to the right locations in the global vectors and an MPI_Allreduce to collectively update the change from other processes.

Algorithm 1: LU factorization and forward solve on MPI process p

```
Require: Matrix portion $Z$ $(M_p \times (N_p + N_p^{rhs}))$
MPI process $p$ owns row set $r_p$ and column set $c_p$
// number of columns saved for update
$colcnt = 0$
for $j = 1$ to $N$ do
    // Step 1: Find pivot
    if $j \in c_p$ then
        $s^p \leftarrow KokkosBlas :: iamax(Z_{i \in r_p, j})$
        $\gamma^p \leftarrow Z_{s^p}, j$
        Exchange to compute $\gamma \leftarrow max_p \gamma^p$
        $s \leftarrow$ row index containing the entry $\gamma$
    // Step 2: Generate column update vector $v$ from column $j$ of $Z$
    if $j \in c_p$ then
        $KokkosBlas :: scal(Z_{i \in r_p, j}, 1/\gamma)$
        if $j \in r_p$ then
            $Z_{j,j} = Z_{j,j} * \gamma$// Restore diagonal
        if $s \in r_p$ then
            $Z_{s,j} = Z_{s,j} * \gamma$// Restore diagonal
        Copy $Z_{r_p,j}$ to $v_{r_p,colcnt}$
        Send column $v_{r_p,colcnt}$ and $s$ to processes sharing row set $r_p$
    else
        Receive $s$
    // Step 3: Exchange pivot row and diagonal row, and broadcast
       pivot row
    if $j \in r_p$ then
        Copy $[Z_{j,c_p}, v_{j,1:colcnt}]$ to $w2$
    if $s \in r_p$ then
        $KokkosBlas :: gemm(v_{s,1:colcnt}, u_{1:colcnt,c_p}, Z_{s,c_p})$
        Copy $[Z_{s,c_p}, v_{s,1:colcnt}]$ to $w3$
        Broadcast $w3$ to processes sharing column set $c_p$
        Copy $w3$ to $u_{s,c_p}$
    else
        Receive $w3$ and copy to $u_{s,c_p}$
    if $j \in r_p$ then
        Send $w2$ to pivot owner
    if $s \in r_p$ then
        Receive $w2$ and copy to $[Z_{s,c_p}, v_{s,1:colcnt}]$
    if $j \in r_p$ then
        Copy $w3$ to $[Z_{j,c_p}, v_{j,1:colcnt}]$
    if $j \notin c_p$ then
        Receive $v_{r_p,colcnt}$
    Remove $j$ from $r_p$ and from $c_p$
    $colcnt++$
    // Step 4: Column update and outer product update
    $KokkosBlas :: gemm(v_{r_p,j}, u_{s,1:colcnt}, Z_{r_p,1:colcnt})$
    if $colcnt = BLKSZ$ then
        $KokkosBlas :: gemm(v_{r_p,1:colcnt}, u_{1:colcnt,c_p}, Z_{r_p,c_p})$
```

Algorithm 2: Backward Solve on MPI process p

```
Require: Matrix portion $Z$ $(M_p \times (N_p + N_p^{rhs}))$
MPI process $p$ owns row set $r_p$
for $j = N$ downto $1$ do
    if $j \in r_p$ then
        // Do an elimination step on the column and the rhs owned by
           process $p$
        for $k = 1$ to $N_p^{rhs}$ do
            $u1(k) \leftarrow Z_{j,N_p+k} / Z_{j,j}$
            $Z_{j,N_p+k} \leftarrow u1(k)$
        Broadcast $u1$ in the column communicator
        // Update rhs
        $KokkosBlas :: gemm(Z_{r_p,j}, u1(:, N_p^{rhs}), Z_{r_p,N_p^{rhs}})$
    Send rhs to the processes on the left
    Receive rhs from the processes on the right
```

6 Results

6.1 Experimental Setup

We use the second and the third the fastest supercomputers in the world at the time of this writing for all our experiments, namely the Summit system at the Oak Ridge Leadership Computing Facility (OLCF), and the Sierra system at the Lawrence Livermore National Laboratory.

The Summit system contains 256 racks, each with eighteen IBM POWER9 AC922 nodes, for a total of 4,608 nodes. Each node contains two POWER9 CPUs, twenty two cores each, and six V100 GPUs. Each node has 512GB of DDR4 memory. Each GPU has 16GB of HBM2 memory. The processors within a node are connected by NVIDIA's NVLink 2.0 interconnect. Each link has a peak bandwidth of 25 GB/s (in each direction). The nodes are connected with a Mellanox dual-rail enhanced data rate (EDR) InfiniBand network. The software environment used for the experiments on Summit includes GNU Compiler Collection (GCC) 7.4.0, CUDA 10.1.243, Engineering Scientific Subroutine Library (ESSL) 6.2.0, Spectrum MPI 10.3.1.

The Sierra system has 240 racks, each with eighteen IBM POWER9 AC922 nodes, for a total of 4,320 nodes. Each node contains two POWER9 CPUs, twenty two cores each, and four V100 GPUs. Each node has 256GB of DDR4 memory. Each GPU has 16GB of HBM2 memory. The processors within a node are connected by NVIDIA's NVLink 2.0 interconnect. The nodes are connected with a Mellanox dual-rail enhanced data rate (EDR) InfiniBand network. The software environment used for the experiments on Sierra includes GNU Compiler Collection (GCC) 7.2.1, CUDA 10.1.243, Engineering Scientific Subroutine Library (ESSL) 6.2.0, Spectrum MPI 10.3.0.

In the next two sections, we demonstrate the performance of ADELUS. First, we investigate the performance of ADELUS solving matrices that are randomly generated on the Summit system. This is reasonable as the performance of the solver is not very different based on the values. The pivoting is the only part that could get affected. Random matrices always require pivoting, making this a good test. Second, we integrate ADELUS into a production application code, EIGER, and demonstrate performance on the linear systems from the electromagnetic application on the Sierra system.

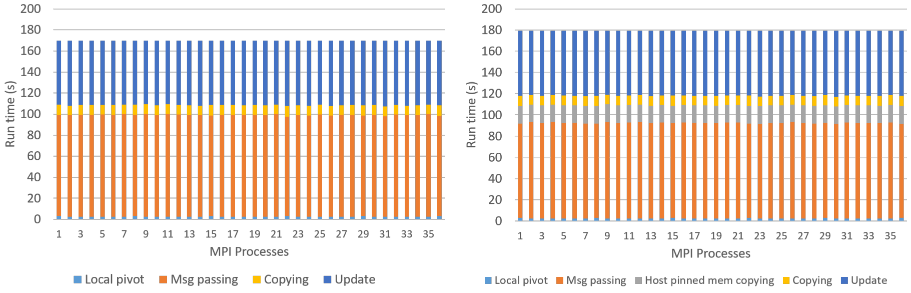
6.2 Performance Results with Randomly-Generated Matrices

In our performance analysis, we run experiments to solve for a linear equation system with a single RHS vector and the matrix size is increased as we increase the hardware resource[1]. Note that the single right hand side problem is typically harder than multiple right hand side problem as there is one less dimension to exploit the parallelism. For the GPU backend, ADELUS runs with one MPI rank per GPU. For the CPU backend, there are three possible MPI rank configurations on the Summit system: (a) 1 MPI rank per node (42 cores each), 1 MPI rank per sockets (21 cores each), or 6 MPI ranks per node (7 cores each). It should be noted that the CPU computation time, which heavily depends on BLAS operations (in which matrix-matrix multiply for the matrix updates is the most time-consuming), dominates the total CPU execution time, as compared to the communication time. We observe that the best performance for CPU execution is reached by assigning all 42 cores for 1 MPI rank. Consequently, in our experiments, ADELUS runs with one 42-core CPU node per MPI process on CPU backend. Since the CPU memory capacity is much larger than the GPU memory capacity, it is difficult to determine a fair comparison scheme between the two backends. In this study, we opt to use the memory occupied by a matrix ($N \times N$) represented in double complex precision in a single GPU as the baseline. As the number of MPI processes increases, the problem (i.e. matrix) sizes are increased so that each MPI process holds the same amount of matrix portion ($N \times N$). The baseline $N \times N$ matrix is chosen with $N = 27{,}882$ which takes 77.7% of 16GB GPU memory. The matrix sizes will be $N \times N$, $2N \times 2N$, ... $\sqrt{p}N \times \sqrt{p}N$, where p is the number processes, in the 1, 4 (2 processes/row), ... p processes ($\sqrt{p}\ processes/row$), respectively. It is noted that ADELUS can handle non-square matrix portion in MPI processes. In Section 6.3, we will show the results of different matrix distributions. For the GPU backend, we test MPI data buffers allocated in GPU memory (CUDA-aware MPI) and host pinned memory.

Load Balancing Verification We first look at the execution time on all MPI processes by picking the matrix size of $6N \times 6N$ running on 36 GPUs. Fig. 2a and Fig. 2b show the timing breakdowns for each of the 36 processes (36 GPUs) for the factorization step in solving the $167,292 \times 167,292$ problem in double complex precision using CUDA-aware MPI and host pinned memory, respectively. The

[1] The driver code used for our ADELUS experiments can be found in https://github.com/trilinos/Trilinos/tree/master/packages/adelus/example

timing breakdown includes the time to find the local maximum entries (called *Local pivot*), the time for MPI communication (called *Msg passing*), the time for internal copying (called *Copying*), and the time for updating matrix (called *Update*). In case of using host pinned memory for MPI, the time for copying back and forth between the device memory and the host pinned memory is included (called *Host pinned mem copying*). It is observed that the workload (computation and communication) is almost perfectly balanced across all the MPI processes while the process idle time is kept minimized due to the torus-wrap mapping scheme. When host pinned memory is used for MPI communication, extra memory copying is explicitly made which results in the increase in the total time. We observe that the communication and the update contribute the most to the total time and the communication time is even higher than the update time (1.47x-1.6x) with this certain problem size on 36 MPI processes. This ratio is expected to increase as more nodes are added. More analysis of the communication and computation is provided in the following sections.

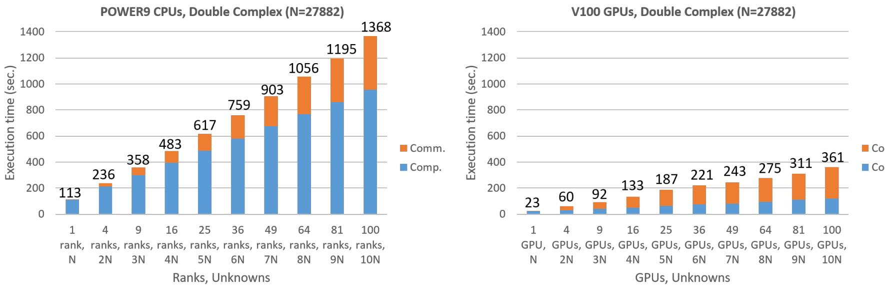


(a) (b)

Fig. 2: Timing breakdowns of the factorization for the $6N \times 6N$ problem on 36 MPI processes using: (a) Cuda-aware MPI; (b) host pinned memory for MPI.

CPU vs. GPU Performance Comparisons The CPU and GPU (using host pinned memory for MPI) computation time and communication time where the problem size varies from $N \times N$ on 1 MPI rank to $10N \times 10N$ matrix using 100 MPI ranks are shown in Fig. 3a and Fig. 3b, respectively. The computation time is defined by subtracting the overhead associated with MPI communication from the total execution time. We can make several observations. First, when a single GPU is compared to 42 cores of the CPU we see a speedup of 4.9 (23 seconds vs 113 seconds). Second, the GPU times increase from 23 seconds to 361 seconds from 1 rank to 100 ranks as the problem size grows one hundred times while the FLOPS grow O(N^3). For the same increase in problem size, the CPU times increase from 113 to 1368 from 1 rank to 100 ranks. Finally, we can see that the GPU total execution for the $10N \times 10N$ problem on 100 processes outperforms the CPU total execution with a speedup factor of 3.8. The ratios between communication and computation are 0.43 (CPU) and 2 (GPU) for the $10N \times 10N$

problem. As processing larger problems (by more MPI processes), communication overhead increases. This communication overhead is mostly contributed by the cost of broadcasting pivot rows (Line 24 of Algorithm 1–Factorization and Forward solve) and the cost of exchanging rhs vectors to left and right processes (Line 9 and 10 of Algorithm 2–Backward solve). It is noted that messages sizes depend on the size of the matrix portion held by MPI processes and these two communication happen at each iteration of the algorithms. In spite of that, CPU computation is still the dominant component in the total CPU time. However, in GPU computation, due to the fact that the computation cost is reduced by the increased parallelism on the GPUs, the communication overhead now becomes the bottleneck.

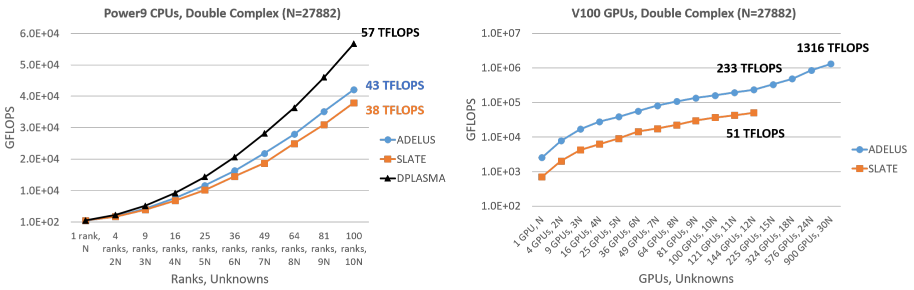


(a) (b)

Fig. 3: ADELUS execution times (double complex precision): (a) CPU execution times. The total CPU time at $10N$x$10N$ is 1368s; (b) GPU execution times with host pinned memory. The total GPU time at $10N$x$10N$ is 361s.

Performance Comparison with DPLASMA and SLATE ADELUS is compared against the two state-of-the-art solver packages DPLASMA [20] (CPU runs) and SLATE [25] (CPU and GPU runs) on the Summit system using the *GESV* testing programs accompanied with the packages . It should be highlighted that IBM XL C/C++ Compiler 16.1.1 is used to build DPLASMA, instead of GCC 7.4.0. For building SLATE, we use GCC 6.4.0 and ESSL 6.1.0, Netlib SCALAPACK 2.0.2. DPLASMA's and SLATE's testing programs have multiple tuning parameters. We identify the values of these parameters that could give the best performance on CPUs and GPUs. We do not use the default parameters for these third party libraries. We tune them to obtain the best performance out of them. We also compare against DPLASMA despite it having the option to do only incremental pivoting while ADELUS does partial pivoting. More specifically, for DPLASMA with *GESV* functionality on CPUs, a square tile with size of 352 is exploited. For SLATE on CPUs, we can achieve the best performance with nb = 320, ib = 32, $panel\_threads$ = 4. For SLATE's *GESV* runs on GPUs, the best performance can be obtained with nb = 640, ib = 32, $panel\_threads$ = 1. Fig. 4a gives GFLOPS performance of the three packages solving up to a $10N$x$10N$ matrix with 100 MPI processes on CPUs. The CPU performance of ADELUS is higher than the CPU performance of SLATE (43 TFLOPS vs. 38 TFLOPS). This can be explained by the fact that SLATE uses OpenMP threads explicitly for multitasking on individual tiles and uses BLAS functions in sequential mode while ADELUS uses multi-threaded BLAS routines. DPLASMA, with its use of the PaRSEC runtime to overlap computation and communication and to dynamically manage and schedule tasks, outperforms ADELUS on CPUs (57 TFLOPS vs. 43 TFLOPS). However, it is noted that DPLASMA does not provide the *GESV* testing with partial pivoting. We use the incremental pivoting for DPLASMA runs instead.

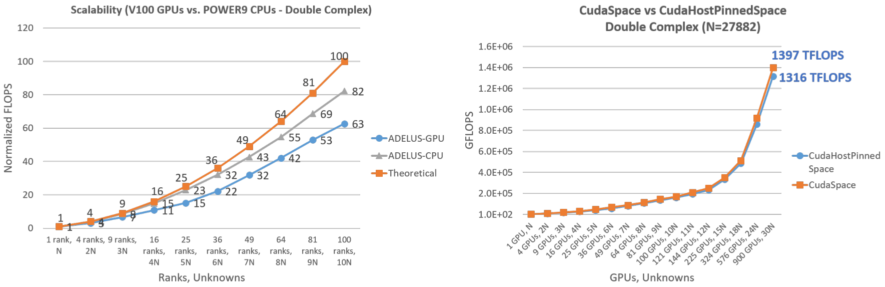
The GPU performance comparison is given in Fig. 4b. Due to the job time limit on Summit, we could not run SLATE further than 144 GPUs solving for $12N$x$12N$ matrix. As we can see, ADELUS delivers superior performance compared to SLATE. Using 144 GPUs, ADELUS can be 4.57x faster than SLATE. Two possible reasons are the use of batched BLAS calls on batches of tiles in SLATE and extra complication of layout translation for row swapping operation in SLATE's GPU acceleration. Another possible reason for the inferior performance of SLATE could be the overhead of simultaneous OpenmP tasks issuing MPI communications during the panel factorization in the SLATE's LU implementation. ADELUS can achieve 1,316 TFLOPS (1.3 PFLOPS) when running on 900 GPUs. To the best of our knowledge, this is the first time that a complex, dense LU solver can reach PFLOPS performance. We also emphasize that ADELUS code is identical for the CPU and GPU evaluations except one template parameter and any use of host pinned memory for MPI communication.



(a) (b)

Fig. 4: GFLOPS (double complex precision): (a) ADELUS vs. DPLASMA and SLATE on Power9 CPUs; (b) ADELUS vs. SLATE on V100 GPUs.

Scalability Analysis In order to investigate the scalability of ADELUS, we compare how the GFLOPS performance improves with more GPUs or more nodes while we increase the matrix size, as shown in Fig. 5a. Scalability is defined as the normalized GFLOPS performance of multiple MPI processes in reference to GFLOPS performance of a single MPI process. In general, the increase of communication overhead results in less than ideal scalability in both CPU and GPU runs. It can be seen that ADELUS running on CPUs scales more closely to the theoretical ideal scalability than ADELUS running on GPUs. This can be explained by the increase in the communication costs on GPUs. This also demonstrates the fact that ADELUS clearly benefits from GPU acceleration. However, notice that the GPU's single GPU GFLOPS was already quite high, so the increase in communication cost shows in the scaling plots.



(a) (b)

Fig. 5: ADELUS (a) Scalability (double complex precision, host pinned memory for MPI is used with GPU backend); (b) CUDA backend execution - using Cuda Host Pinned Memory vs Cuda Memory for MPI.

MPI Buffers on Different Memory Spaces ADELUS has an option which allows one to choose whether using host pinned memory as MPI buffers or use CUDA-aware MPI during the communication. Fig. 5b shows the GFLOPS performance of the GPU execution with respect to the increase of problem size. Both memory spaces, namely CudaSpace and CudaHostPinnedSpace, can attain performance above 1000 TFLOPs. Using CUDA-aware MPI can improve the performance by 6% since we do not need to explicitly buffer data on host memory before or after calling the MPI function.

6.3 Performance Results from Large-Scale EM Simulation

We demonstrate ADELUS performance on a real computational science application on 100 GPUs to 7600 GPUs by integrating it with the electromagnetics simulation code EIGER (written in Fortran). Several numerical simulations were performed on the Sierra platform available at the LLNL using EIGER coupled with the ADELUS solver. The performance results are shown in Table 1. The NVIDIA GPUs were used in the solve and since there are 4 GPUs per node, the number of MPI processes is four times the number of nodes.

A number of observations can be made from Table 1. First, the performance of the solver increases with the number of nodes. *ADELUS reaches 7.72 Petaflops when using 7600 GPUs.* This translates to 16.9% of theoretical double precision

Table 1: ADELUS Solver Performance on Large Scale EM Simulations. Nodes are shown. Number of GPUs are four times the number of nodes.

Order(N)	Nodes	Solve Time(s)	TFLOPS	Procs/Row (P_c)
226,647	25	240.5	1291.	10
1,065,761	310	1905.1	1694.5	31
1,322,920	500	6443.9	958.1	20
1,322,920	500	2300.2	2684.1	50
1,322,920	500	2063.6	2991.9	100
2,002,566	1200	3544.1	6042.6	100
2,564,487	1900	5825.2	7720.7	80

floating point performance if we only account for computation cost in theory. In addition, the performance is affected by the distribution of the matrix on the MPI processes. This is revealed by the 1.3 million unknown problem where assigning more processes per row yields higher performance. We hypothesize this is due to the reduction of communication cost of broadcasting pivot rows during partial pivoting (Line 24 of Algorithm 1). However, the overhead of communicating rhs vectors to left and right processes (Line 9 and 10 of Algorithm 2) also contributes to the total performance. As we have more processes per row, this communication overhead in the backward solve increases. Therefore, we observe the performance improvement of 1.1x when going from 50 processes/row to 100 processes/row (as compared to 2.8x going from 20 processes/row to 50 processes/row) in Table 1. The selection of the number of processes per row P_c (and the number of processes per column P_r) for best performance is heuristic-based and should be a compromise to both the aforementioned communication overheads. It is common to choose $P_c = P_r$ or P_c slightly greater than P_r for an acceptably good performance. Not shown in Table 1 is the per process performance and for the problems and distributions used has a maximum value of 1.5 Tflops/rank.

7 Conclusions and Future Work

In this paper, we present a parallel, dense, performance-portable, LU solver based on torus-wrap mapping and LU factorization algorithm. Using the portability provided by Kokkos, the solver can be portable to CPUs and GPUs. The performance evaluation of ADELUS is demonstrated on the Summit system, in which it achieves 1.397 PFLOPS on 900 GPUs. It is shown that, the GPU execution outperforms the CPU execution (with 42 cores) in terms of speedup by a factor of 3.8. We also demonstrate the integration of the ADELUS solver into an electromagnetic application achieving a performance of 7.720 PFLOPS on 7600 GPUs when solving a problem of 2.5M unknowns on the Sierra system. ADELUS scalability on the GPU backend could be resolved by exploiting more computation-communication overlapping techniques. Another issue that remains to be resolved is the limitation of the GPU memory. Since ADELUS execution is exclusive to one memory space, when the problem size exceeds the GPU memory limit, more GPUs need to be accommodated. One possible solution to overcome this limitation is a hybrid implementation where both CPU and GPU resources are fully utilized. Our future investigation would address these issues.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. Gross, E., Harrington, H.A., Rosen, Z., Sturmfels, B.: Algebraic systems biology: A case study for the Wnt pathway. Bulletin of Mathematical Biology **78**(1), 21–51 (2016)
2. Judd, K.L.: Numerical methods in economics. MIT Press, Cambridge, MA (1998)
3. Larson, R.: Elementary linear algebra. 8th edn. Cengage Learning, Boston, MA (2017)
4. Wrobel, L.C., Aliabadi M.H.: The boundary element method, volume 1: Applications in thermo-fluids and acoustics. Wiley, New York (2002)
5. Wrobel, L.C., Aliabadi M.H.: The boundary element method, volume 2: Applications in solids and structures. Wiley, New York (2002)
6. Harrington, R.F.: Field computation by moment method. Wiley-IEEE Press, New York (1993)
7. Bettencourt, M.T., Zinser, B., Jorgenson, R.E., Kotulski, J.D.: Performance portable sparse approximate inverse preconditioner for EFIE equations. In: Proceedings of the International Conference on Electromagnetics in Advanced Applications (ICEAA), pp. 1469–1472. IEEE, Verona (2017)
8. Coifman, R., Rokhlin, V., Wandzura, S.: The fast multipole method for the wave equation: A pedestrian prescription. IEEE Antennas Propag. Mag. **35**(3), 7–12 (1993)
9. Song, J.M., Chew, W.C.: Multilevel fast multipole algorithm for solving combined field integral equations of electromagnetic scattering. Microw. Opt. Technol. Lett. **10**, 14–19 (1995)
10. Oak Ridge Leadership Computing Facility, https://www.olcf.ornl.gov/summit/. Last accessed on 20 Apr 2020
11. Livermore Computing Center - High Performance Computing, https://hpc.llnl.gov/hardware/platforms/sierra/. Last accessed on 23 May 2020
12. Argonne Leadership Computing Facility, https://aurora.alcf.anl.gov/. Last accessed on 20 Apr 2020
13. Oak Ridge Leadership Computing Facility, https://www.olcf.ornl.gov/frontier/. Last accessed on 20 Apr 2020

14. Smith, R.: El Capitan supercomputer detailed: AMD CPUs & GPUs to drive 2 exaflops of compute. AnandTech (Mar. 2020). https://www.anandtech.com/show/15581/el-capitan-supercomputer-detailed-amd-cpus-gpus-2-exaflops
15. Hendrickson, B.A., Womble, D.E.: The torus-wrap mapping for dense matrix calculations on massively parallel computers. SIAM J. Sci. Comput. **15**(5), 1201–1226 (1994)
16. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comp. **74**(12), 3202–3216 (2014)
17. Kokkos Kernels, https://github.com/kokkos/kokkos-kernels. Last accessed on 26 Aug 2020
18. Blackford, L.S., et al.: ScaLAPACK Users' Guide. SIAM, Philadelphia, PA (1997)
19. Poulson, J., Marker, B., Van de Geijn, R.A., Hammond, J.R., Romero, N.A.: Elemental: A new framework for distributed memory dense matrix computations. ACM T. Math. Software (TOMS) **39**(2), 13 (2013)
20. Bosilca, G., et al.: Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1432–1441. IEEE, Shanghai (2011). https://bitbucket.org/icldistcomp/dplasma
21. Bosilca, G., et al.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing **38**(1–2), 37–51 (2012)
22. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: Hybrid GPU accelerated linear algebra routines. In: Proceedings of SPIE Defense and Security Symposium (DSS) (2010)
23. Dongarra, J., et al.: Accelerating numerical dense linear algebra calculations with GPUs. In: Kindratenko, V. (eds) Numerical computations with GPUs. Springer, Cham (2014)
24. cuSOLVER library, https://docs.nvidia.com/cuda/cusolver. Last accessed on 26 Aug 2020
25. Gates, M., et al.: SLATE: design of a modern distributed and accelerated linear algebra library. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–18. ACM, Denver (2019). https://bitbucket.org/icl/slate
26. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
27. Farber, R.: Parallel programming with OpenACC. Morgan Kaufmann Publishers Inc., San Francisco, CA (2016)
28. Beckingsale, D.A., et al.: RAJA: Portable performance for large-scale scientific applications. In: Proceedings of 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81. IEEE, Denver (2019). https://github.com/LLNL/RAJA
29. Rao, S.M., Wilton, D.R., Glisson, A.W.: Electromagnetic scattering by surface of arbitrary shape. IEEE Trans. on Antennas and Propagat. **30**(3), 409–418 (1982)
30. Wilton, D.R., et al.: EIGER: A new generation of computational electromagnetics tools. In: Proceedings of ElectroSft: Software for EE Analysis and Design, pp. 28–30, Miniato, Italy (1996)
31. W. L. Langston, et al. Massively parallel frequency domain electromagnetic simulation codes. In: Proceedings of International Applied Computational Electromagnetics Society Symposium (ACES), Denver, CO (2018)