SAND2020-5338C

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

Vinh Q. Dang
Dept of EM Theory & Simulation
Sandia National Laboratories
Albuquerque, NM, 87123, USA
vqdang@sandia.gov

Joseph D. Kotulski
Dept of EM Theory & Simulation
Sandia National Laboratories
Albuquerque, NM, 87123, USA
jdkotul@sandia.gov

Sivasankaran Rajamanickam
Dept of Scalable Algorithms
Sandia National Laboratories
Albuquerque, NM, 87123, USA
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 ADELUS (A Dense LU Solver) package for current and next generation distributed hardware-accelerated high-performance computing platforms. The package performs solves of dense linear systems through 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. The 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 are provided. Preliminary performance results from large-scale electromagnetic simulations using the dense LU solver are also presented.**

***Index Terms*—dense linear systems of equations, distributed computing, GPU acceleration, LU factorization, performance portability**

I. 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 including acoustics, electrochemistry, fluid mechanics [4], elastodynamics, fracture mechanics [5], electromagnetics (method of moments) [6], etc. 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. However, 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 heavily computational burden of direct solvers, one can seek for 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 accurate results in cases where many iterative solvers fail to solve correctly 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 direct solvers using distributed-memory hardware-accelerated computing platforms to mitigate their aforementioned burden and allow them to target extremely large-scale problems.

Nowadays, most high-performance computing (HPC) platforms are heterogeneous machines that are equipped with central processing units (CPUs) and hardware-based accelerators like graphical processing units (GPUs). Nearly 40 percent of the total compute power on the TOP500 list comes from GPU-accelerated systems [10]. The current fastest machine on the TOP500 list is the Summit system [11] located at the Oak Ridge National Laboratory (ORNL). The Summit system contains six NVIDIA V100 GPUs and two POWER9 CPUs per node. The peak double-precision floating-point performance of the CPU is 44 (cores) × 24.56 GFLOPS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA0003525.

= 1.08 TFLOPS. The peak performance of the GPUs is 6 (devices) × 7.8 TFLOPS = 46.8 TFLOPS. Recently, the U.S. Department of Energy has announced plans for three exascale-class supercomputers: (1) Aurora system [12], built 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 arch-based GPUs; (2) Frontier system [13] will be built at ORNL. It is planned to debut in 2021 and deliver 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] is scheduled to be installed in the Lawrence Livermore National Laboratory (LLNL) in early 2023 and to deliver 2 exaFLOPS of theoretical peak performance. Each El Capitan node will contain one AMD EPYC CPU, code-named "Genoa" and featuring the "Zen 4" processor core, and four next-generation AMD Radeon Instinct GPUs. Next generation exascale HPC architectures are continuously evolving to allow for larger, more computationally intensive problems to be solved. At the same time, they have introduced new challenges to algorithm designs and implementations due to the significantly different architectures and programming models. Therefore, it is important to design flexible algorithms and flexible implementations that can perform well (i.e. performance-portable) on various platforms.

This paper presents ADELUS, a performance-portable dense LU solver for current and next generation distributed-memory hardware-accelerated HPC platforms. ADELUS performs LU factorization with partial pivoting and solves dense linear equation systems using 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 perform factorization and solve their own tasks 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 KokkosKernels 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 LU factorization algorithm for traditional and accelerator-based architectures that can achieved 1397 TFLOPS on 900 GPUs. 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 7720 TFLOPS.

The rest of the paper is organized such that Section II describes related work. Section III provides an overview of Kokkos and Kokkos Kernels followed by an overview of the method of moments in Section IV. Section V describes the implementation of the solver on distributed-memory clusters. Experimental results are discussed in Section VI. Finally, our findings are summarized in Section VII.

II. Related Work

There have been many related research efforts in the literature which makes it infeasible to survey all of them. In this section, we hence list the most popular software packages which implement LU solvers related to distributed memory and/or GPU accelerators. Distributed-memory implementations are available in:

- ScaLAPACK [18]: ScaLAPACK can be considered as 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 modern 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. Elemental has not been maintained since 2016. However, the project was forked by the LLNL under the name Hydrogen, to make use of GPU accelerators, required by the Livermore Big Artificial Neural Network toolkit. The supported functionality is only limited to the basic utilities and BLAS-1,-3 operations;
- DPLASMA [20]: 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 GPU acceleration for LU solver. Moreover, DPLASMA does not support 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 that addresses performance portability for LU solver via Kokkos/Kokkos Kernels libraries on distributed-memory hardware-accelrated machines.

III. 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 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 compile time template parameters. There are backends available for OpenMP, CUDA for NVIDIA GPUs, and HIP for AMD 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 View 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 the 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.

IV. Application: Method of Moments for Linear Electromagnetics

An important class of problems that can be solved with the ADELUS solver are those encountered in 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]. These equations are then tested using the Galerkin approach to produce a complex, dense, double-precision matrix. 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. Note that the discretization required to solve problems of interest forces the usage of capability machines that are efficient in both message passing (MPI), threading on advanced architectures (GPU'S), and computational performance. To this end ADELUS has been successfully integrated with the method of moments code EIGER [29]. This production 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 [30], is currently being developed to use the Kokkos library to increase performance in the filling of the matrix as well.

V. Parallel LU Solver Implementation

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

A. ADELUS Interface and Storage

The current ADELUS solver requires the matrix and RHS vectors are packed together and computed before ADELUS is called since the forward solve is realized by factoring 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.

As viewed by a user, the matrix is block-mapped to the MPI processes. The matrix is distributed to the MPI processes using the criterion: the maximum difference in the number of rows (or columns) assigned to MPI processes is one. The same rule is applied to the RHS vectors. ADELUS provides a distribution utility function for users to calculate the workload on each MPI process. 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 block map. Fig. 1 shows an example of mapping the original

matrix and 2 RHS vectors to 6 MPI processes with 3 processes per row. This information is used by user code to construct portions of the matrix and RHS vectors correctly on each MPI process. ADELUS is then called by MPI processes taking the portions of matrix packed with RHS vectors as their inputs.

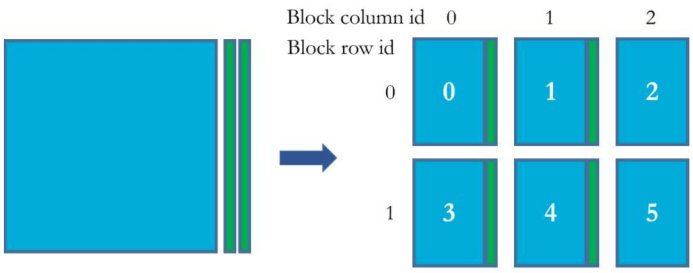

Fig. 1. An example of ADELUS workload distribution for 6 MPI processes, 3 processes in a row, and 2 RHS vectors. The MPI process indices are shown in the boxes.

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. 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.). For example, one can allocate a view 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);
```

In the current version of ADELUS, the implementation is exclusive, 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.

B. 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 x P_c, one can construct a block mapping with the block sizes of M_p x 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 process 0, columns 2, P_c+2, $2P_c$+2, ... to process 1, etc. For rows, ADELUS assigns rows 1, P_r+1, $2P_r$+1, ... to process P_c, rows 2, P_r+2, $2P_r$+2, ... to process $2P_c$, etc. 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. As stated in [15], it is not necessary to redistribute the block-mapped matrix among processes for torus-wrapped solver. 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. 2 shows an example of torus-mapping matrix elements to 6 MPI processes with 3 processes per row.

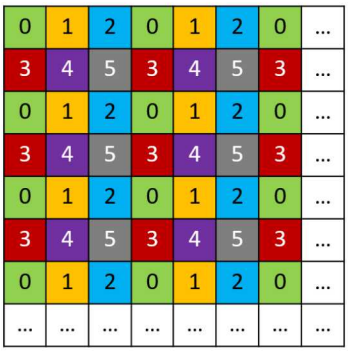

Fig. 2. An example of ADELUS torus-wrap mapping for 6 MPI processes, 3 processes in a row. The MPI process indices are displayed at each matrix element.

C. LU Solver

In ADELUS, the LU solver comprises three main steps: LU factorization+forward solve, backward solve, and permutation.

1) LU Factorization and Forward Solve: Since the forward solve is performed in a similar way to the LU factorization, we merge the forward solve with the factorization for coding simplicity. We implement the conventional right-looking variant of the LU factorization with partial pivoting of a dense N x N matrix. The algorithm is summarized in Algorithm 1.

Each MPI process handles its own workload through 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 third-party library routines optimized for multi-threaded CPU and massively parallel GPU architectures. Depending on where the data resides in, 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, 24, 26, 28, 32, and 34), (iv) *KokkosBlas::gemm* for updating the matrix (Lines 23, 37, and 39). Our algorithm requires only simple

communication patterns consisting of point-to-point communication: MPI_Send, MPI_Recv, MPI_Irecv (Lines 16, 18, 30, and 32 of Algorithm 1) and collective communication: MPI_Bcast, MPI_Allreduce (Lines 7 and 25). 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 CUDA-aware MPI.

We employ the delay-updating technique (Line 38 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 are 96 while a typical value of BLKSZ for GPU backend are 128. These numbers are used in our performance evaluation in Section VI. The algorithm utilizes overlapping technique which performs column updates within a block one column at a time. To minimize the waiting time, the algorithm attempts to do row work while waiting for column to arrive.

2) Backward Solve: In the backward solve 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.

3) Permutation: Since the torus-wrapping scheme is assumed by the solver while the input matrix is not torus-wrapped, permutation of the solution vectors must be carried out to "unwrap the results". Each process that owns local solution vectors creates a temporary buffer for global solution vectors. The permutation simply involves a Kokkos $parallel\_for$ 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.

VI. Results

A. Experimental Setup

We perform our experimental work using two computing systems: (1) the Summit system at the Oak Ridge Leadership Computing Facility (OLCF), and (2) the Sierra system at the Lawrence Livermore National Laboratory.

(1) 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.

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

Require: Matrix portion Z (M_px(N_p+N_p^{rhs}))
1 MPI process p owns row set r_p and column set c_p
`// number of columns saved for update`
2 $colcnt = 0$
3 **for** $j = 1$ **to** N **do**
`// Find pivot`
4 **if** $j \in c_p$ **then**
`// Find maximum of entries in column` j
5 $s^p \leftarrow KokkosBlas :: iamax(Z_{i \in r_p, j})$
6 $\gamma^p \leftarrow Z_{s^p}, j$
7 Exchange to compute $\gamma \leftarrow max_p \gamma^p$
8 $s \leftarrow$ row index containing the entry γ
`// Generate column update vector` v `from column` j `of` Z
9 **if** $j \in c_p$ **then**
10 $KokkosBlas :: scal(Z_{i \in r_p, j}, 1/\gamma)$
11 **if** $j \in r_p$ **then**
12 $Z_{j,j} = Z_{j,j} * \gamma$`// Restore diagonal`
13 **if** $s \in r_p$ **then**
14 $Z_{s,j} = Z_{s,j} * \gamma$`// Restore diagonal`
15 Copy $Z_{r_p,j}$ to $v_{r_p,colcnt}$
16 Send column $v_{r_p,colcnt}$ and s to processes sharing row set r_p
17 **else**
18 Receive $v_{r_p,colcnt}$ and s
`// Exchange pivot row and diagonal row, and broadcast pivot row`
19 **if** $j \in r_p$ **then**
20 Copy $[Z_{j,c_p}, v_{j,1:colcnt}]$ to $w2$
21 **if** $s \in r_p$ **then**
22 **if** $colcnt > 0$ **then**
23 $KokkosBlas :: gemm(v_{s,1:colcnt}, u_{1:colcnt,c_p}, Z_{s,c_p})$
24 Copy $[Z_{s,c_p}, v_{s,1:colcnt}]$ to $w3$
25 Broadcast $w3$ to processes sharing column set c_p
26 Copy $w3$ to u_{s,c_p}
27 **else**
28 Receive $w3$ and copy to u_{s,c_p}
29 **if** $j \in r_p$ **then**
30 Send $w2$ to pivot owner
31 **if** $s \in r_p$ **then**
32 Receive $w2$ and copy to $[Z_{s,c_p}, v_{s,1:colcnt}]$
33 **if** $j \in r_p$ **then**
34 Copy $w3$ to $[Z_{j,c_p}, v_{j,1:colcnt}]$
35 Remove j from r_p and from c_p
36 $colcnt++$
37 $KokkosBlas :: gemm(v_{r_p,j}, u_{s,1:colcnt}, Z_{r_p,1:colcnt})$
38 **if** $colcnt$ = *BLKSZ* **then**
`// saved enough columns`
39 $KokkosBlas :: gemm(v_{r_p,1:colcnt}, u_{1:colcnt,c_p}, Z_{r_p,c_p})$

Algorithm 2: Backward Solve on MPI process p

Require: Matrix portion Z (M_px(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

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.

(2) The Sierra system contains 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.

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 investigate the performance of ADELUS solving random matrices on the Summit system and the application performance when ADELUS is integrated into the electromagnetic application on the Sierra system.

B. 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. For GPU backend, ADELUS runs with one GPU per MPI process while it runs with one 42-core CPU node per MPI process to provide the most threading parallelism for the best performance 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*x*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*x*N*). The baseline *N*x*N* matrix is chosen with N = 27,882 which takes 77.7% of 16GB GPU memory. The matrix sizes will be *N*x*N*, 2*N*x2*N*, 3*N*x3*N*, 4*N*x4*N*, 5*N*x5*N*, 6*N*x6*N*, ... assigned to 1, 4, 9, 16, 25, 36, ... processes, respectively. It is noted that, for GPU backend, both MPI using CUDA memory (CUDA-aware MPI) and MPI using host pinned memory are tested.

1) Load Balancing Verification: We first look at the execution time on all MPI processes by picking the matrix size of 6*N*x6*N* running on 36 GPUs. Fig. 3 and Fig. 4 show the timing breakdowns for each of the 36 processes (36 GPUs) for the factorization step in solving the 167,292x167,292 problem in double complex precision using CUDA-aware MPI and host pinned memory, respectively. The 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 yields 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 growing higher as more nodes are added. More analysis of the communication and computation is provided in the next section.

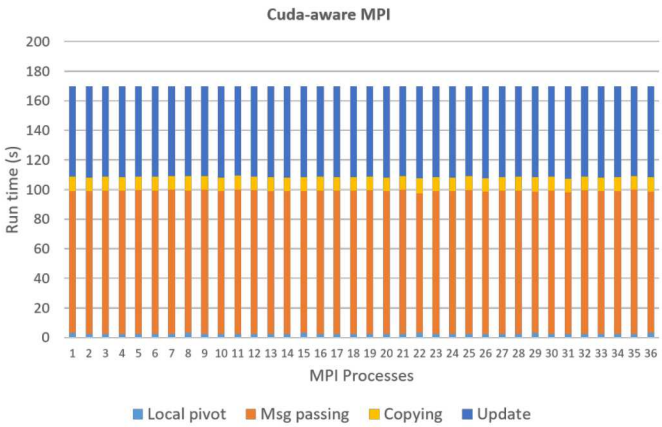

Fig. 3. Load distribution of the factorization for 167292x167292 problem (Cuda-aware MPI).

2) Performance Evaluation–CPU vs. GPU: The CPU and GPU (using host pinned memory for MPI) computation time and communication time for solving up to 10*N*x10*N* matrix using 100 MPI processes are shown in Fig. 5 and Fig. 6, respectively. The computation time is defined by subtracting

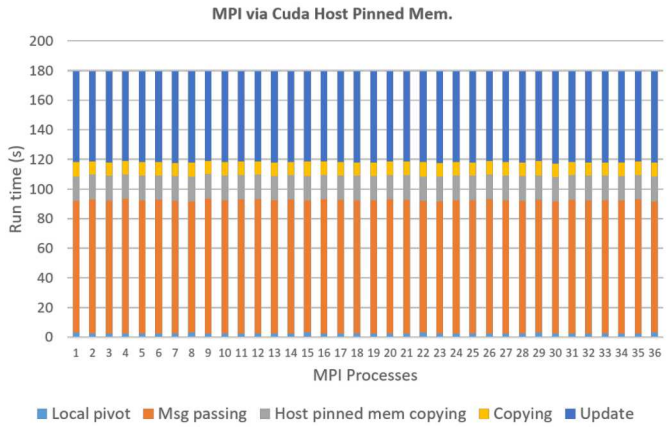

Fig. 4. Load distribution of the factorization for 167292x167292 problem (MPI using Cuda host pinned memory).

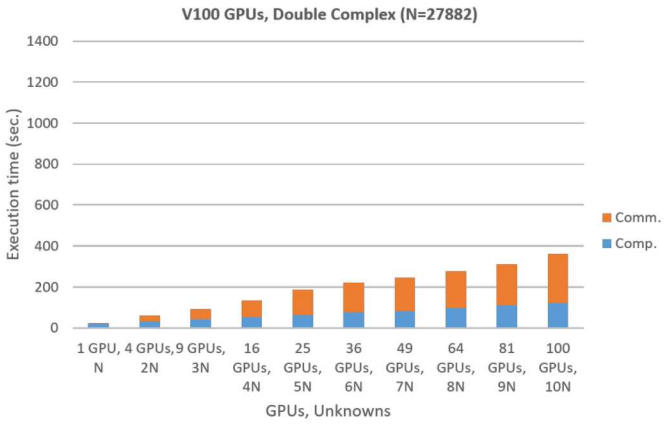

Fig. 6. ADELUS: GPU execution times (double complex precision) with host pinned memory. The total GPU time at 10Nx10N is 361s.

the overhead associated with MPI communication from the total execution time. We can see that the GPU total execution for the 10Nx10N problem on 100 processes outperforms the CPU total execution with a speedup factor of 3.8. The ratios between communication and computation are up to 0.43 (CPU) and 2 (GPU) for the 10Nx10N problem. As processing more workloads (by more MPI processes), communication overhead increases. In spite of that, CPU computation is still the dominant component. However, in GPU computation, since GPU can help accelerate the computation significantly, the communication overhead now becomes the bottleneck.

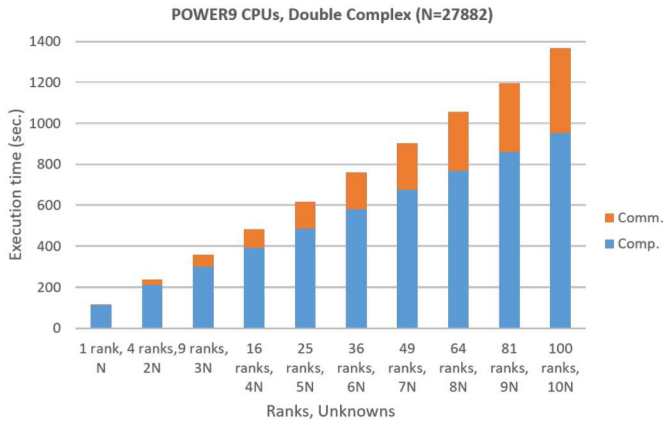

Fig. 5. ADELUS: CPU execution times (double complex precision). The total CPU time at 10Nx10N is 1368s.

3) 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 Netlib LAPACK 3.8.0. 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. 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. 7 gives GFLOPS performance of the three packages solving up to 10Nx10N 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). DPLASMA outperforms ADELUS on CPUs (57 TFLOPS vs. 43 TFLOPS). However, it is noted that DLASMA 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. 8. Due to the job time limit on Summit, we could not run SLATE further than 144 GPUs solving for 12NxN matrix. As we can see, ADELUS delivers superior performance compared to SLATE. Using 144 GPUs, ADELUS can be 4.57x faster than SLATE. 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 dense LU sovler can reach PFLOPS performance.

4) Scalability Analysis: In order to investigate the scalability of ADELUS, we compare how the GFLOPS performance improves with more GPUs (nodes) while we increase the matrix size, as shown in Fig. 9. Scalability is defined as the normalized GFLOPS performance of multiple MPI processes in reference to GFLOPS performance of a single MPI process. The increase of communication overhead results in the relatively poor scalability in both CPU and GPU runs. It can be seen that ADELUS running on CPUs scales more closely to the theoretical linear expectation than ADELUS running on GPUs. This can be explained by the dominant computation time on CPUs and dominant communication time on GPUs. This also demonstrates the fact that ADELUS clearly benefits

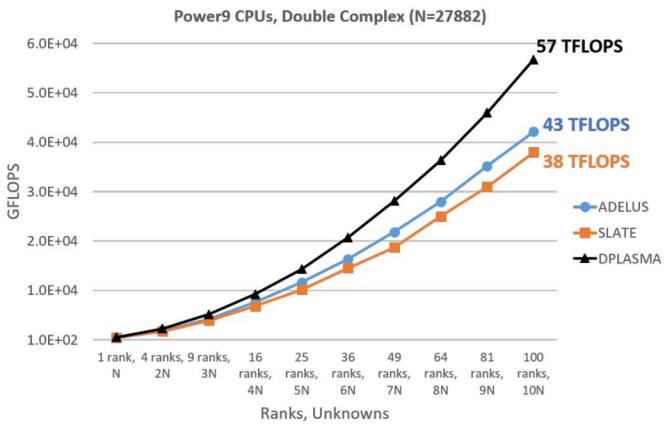

Fig. 7. Performance GFLOPS: ADELUS vs. DPLASMA and SLATE on Power9 CPUs for double complex precision.

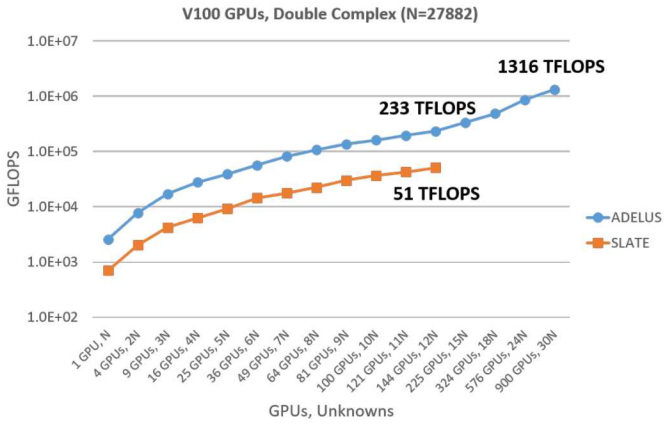

Fig. 8. Performance GFLOPS: ADELUS vs. SLATE on V100 GPUs for double complex precision.

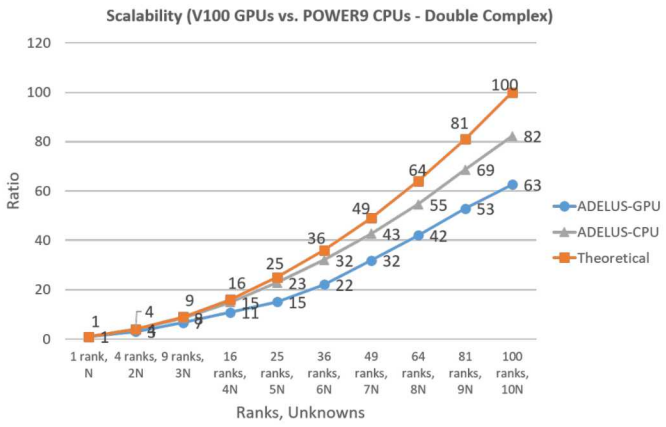

Fig. 9. ADELUS: Scalabilty (double complex precision). Host pinned memory for MPI when GPUs are used.

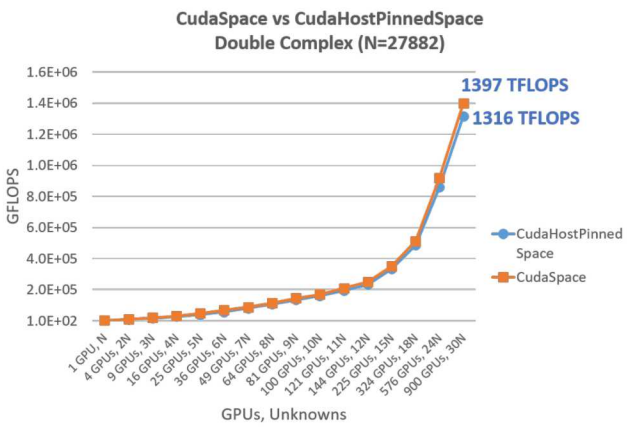

Fig. 10. ADELUS: CUDA backend execution - using Cuda Host Pinned Memory vs Cuda Memory for MPI.

from GPU acceleration.

5) Performance Evaluation with MPI Buffers on Different Memory Spaces: ADELUS has an option which allows one to choose whether using host pinned memory as MPI buffers during the communication. Fig. 10 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 the performance above 1000 TFLOPs. Using CUDA-aware MPI can improve the performance to 6% since we do not need to explicitly buffer data on host memory before or after calling the MPI function.

C. Performance Results from Large-Scale EM Simulation

Several numerical simulations were performed on the SIERRA platform available at Lawrence Livermore National Laboratory using EIGER coupled with the ADELUS solver. The performance results are displayed in Table 1. The NVIDIA GPU's 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.

Table 1. ADELUS Solver Performance on Large Scale EM Simulations.

Order(N)	Nodes	Solve Time(s)	TFLOPS	Procs/Row
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

A number of observations can be made from Table 1. First, the performance of the solver increases with the number of nodes. 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 processors per row yields higher performance. Not shown in the Table is the per process performance and for the problems and distributions used has a maximum value of 1.5 Tflops/rank.

VII. 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 1397 TFLOPS 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 7720 TFLOPS on 7600 GPUs when solving a problem of 2.5M unknowns on the Sierra system. ADELUS currently suffers from the scalability issue, especially on GPU backend, which can 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, 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] E. Gross, H. A. Harrington, Z. Rosen, and B. Sturmfels, "Algebraic systems biology: A case study for the Wnt pathway," Bulletin of Mathematical Biology, vol. 78, no. 1, pp. 21–51, 2016.

[2] K. L. Judd, Numerical Methods in Economics. Cambridge, MA: MIT Press, 1998, pp.55–58.

[3] R. Larson, Elementary Linear Algebra, 8th ed.. Boston, MA: Cengage Learning, 2017.

[4] L. C. Wrobel and M. H. Aliabadi, The Boundary Element Method, Volume 1: Applications in Thermo-Fluids and Acoustics. New York: Wiley, 2002.

[5] L. C. Wrobel and M. H. Aliabadi, The Boundary Element Method, Volume 2: Applications in Solids and Structures. New York: Wiley, 2002.

[6] R. F. Harrington, Field Computation by Moment Method. New York: Wiley-IEEE Press, 1993.

[7] M. T. Bettencourt, B. Zinser, R. E. Jorgenson, and J. D. Kotulski, "Performance portable sparse approximate inverse preconditioner for EFIE equations," International Conference on Electromagnetics in Advanced Applications (ICEAA), Verona, 2017, pp. 1469-1472.

[8] R. Coifman, V. Rokhlin, and S.Wandzura, "The fast multipole method for the wave equation: A pedestrian prescription," IEEE Antennas Propag. Mag., vol. 35, no. 3, pp. 7–12, Jun. 1993.

[9] J. M. Song and W. C. Chew, "Multilevel fast multipole algorithm for solving combined field integral equations of electromagnetic scattering," Microw. Opt. Technol. Lett., vol. 10, pp. 14–19, Sep. 1995.

[10] P. Kharya, "Record 136 NVIDIA GPU-Accelerated Supercomputers Feature in TOP500 Ranking," NVIDIA, Nov. 19, 2019. Accessed on: Apr. 20, 2020. [Online]. Available: https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/

[11] Oak Ridge Leadership Computing Facility. Accessed on: Apr. 20, 2020. [Online]. Available: https://www.olcf.ornl.gov/summit/

[12] Argonne Leadership Computing Facility. Accessed on: Apr. 20, 2020. [Online]. Available: https://aurora.alcf.anl.gov/

[13] Oak Ridge Leadership Computing Facility. Accessed on: Apr. 20, 2020. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[14] R. Smith, "El Capitan supercomputer detailed: AMD CPUs & GPUs to drive 2 exaflops of compute," AnandTech, Mar. 04, 2020. Accessed on: Apr. 20, 2020. [Online]. Available: https://www.anandtech.com/show/15581/el-capitan-supercomputer-detailed-amd-cpus-gpus-2-exaflops

[15] B. A. Hendrickson and D. E. Womble, "The Torus-Wrap mapping for dense matrix calculations on massively parallel computers," SIAM J. Sci. Comput., vol. 15, no. 5, pp. 1201–1226, 1994.

[16] H.C. Edwards, C.R. Trott, and D. Sunderland, "Kokkos: enabling manycore performance portability through polymorphic memory access patterns," J. Parallel Distrib. Comp., vol. 74, no. 12, pp. 3202–3216, 2014.

[17] Kokkos Kernels. Available: https://github.com/kokkos/kokkos-kernels

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, et al., ScaLAPACK Users' Guide. Philadelphia, PA: SIAM, 1997.

[19] J. Poulson, B. Marker, R. A Van de Geijn, J. R. Hammond, and N. A. Romero, 2013. "Elemental: A new framework for distributed memory dense matrix computations," ACM T. Math. Software (TOMS), vol. 39, no. 2, 2013.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, et al., 2011. "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1432–1441, 2011. Available: https://bitbucket.org/icldistcomp/dplasma

[21] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," Parallel Computing, vol. 38, no. 1-2, pp. 37-51, 2012.

[22] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, E. J. Kelmelis,"CULA: Hybrid GPU accelerated linear algebra routines," SPIE Defense and Security Symposium (DSS), April, 2010.

[23] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with GPUs," in Numerical computations with GPUs. Springer, 2014, pp. 1–26.

[24] NVIDIA Corp., cuSOLVER library. NVIDIA Corp., Nov. 2019. Accessed on: May 19, 2020. [Online: https://docs.nvidia.com/cuda/cusolver].

[25] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "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. 2019. Available: https://bitbucket.org/icl/slate

[26] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," IEEE computational science and engineering, vol. 5, no. 1, pp. 46–55, 1998.

[27] R. Farber, Parallel Programming with OpenACC. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2016.

[28] D. A. Beckingsale et al., "RAJA: Portable performance for large-scale scientific applications," 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 2019, pp. 71-81. Available: https://github.com/LLNL/RAJA

[29] D. R. Wilton, W. A. Johnson, R. Jorgenson, R. Sharpe, and J. Rockway, "EIGER: A new generation of computational electromagnetics tools," in ElectroSft: Software for EE Analysis and Design, Miniato, Italy, May 1996, pp. 28-30.

[30] W. L. Langston et al., "Massively parallel frequency domain electromagnetic simulation codes," International Applied Computational Electromagnetics Society Symposium (ACES), Denver, CO, 2018.